ACM 算法模板

### Dinic算法求网络流

#include <cstdio>

#include <string.h>

#include <queue>

using namespace std;

int const inf = 0x3f3f3f3f;

int const MAX = 205;

int n, m;

int c[MAX][MAX], dep[MAX];//dep[MAX]代表当前层数

int bfs(int s, int t)//重新建图，按层次建图

{

queue<int> q;

while(!q.empty())

q.pop();

memset(dep, -1, sizeof(dep));

dep[s] = 0;

q.push(s);

while(!q.empty()){

int u = q.front();

q.pop();

for(int v = 1; v <= m; v++){

if(c[u][v] > 0 && dep[v] == -1){//如果可以到达且还没有访问，可以到达的条件是剩余容量大于0，没有访问的条件是当前层数还未知

dep[v] = dep[u] + 1;

q.push(v);

}

}

}

return dep[t] != -1;

}

int dfs(int u, int mi, int t)//查找路径上的最小流量

{

if(u == t)

return mi;

int tmp;

for(int v = 1; v <= m; v++){

if(c[u][v] > 0 && dep[v] == dep[u] + 1 && (tmp = dfs(v, min(mi, c[u][v]), t))){

c[u][v] -= tmp;

c[v][u] += tmp;

return tmp;

}

}

return 0;

}

int dinic()

{

int ans = 0, tmp;

while(bfs(1, m)){

while(1){

tmp = dfs(1, inf, m);

if(tmp == 0)

break;

ans += tmp;

}

}

return ans;

}

int main()

{

while(~scanf("%d %d", &n, &m)){

memset(c, 0, sizeof(c));

int u, v, w;

while(n--){

scanf("%d %d %d", &u, &v, &w);

c[u][v] += w;

}

printf("%d\n", dinic());

}

return 0;

}

### 弗洛伊德算法

void floyd()

{

for(int k=1;k<=n;k++)

for(int i=1;i<=n;i++)

for(int j=1;j<=n;j++)

mp[i][j]=min(mp[i][j],mp[i][k]+mp[k][j]);

}

### 欧拉回路判定，逆序输出路径

输入文件由几个块组成。 每个街区描述一个城镇。 块中的每一行包含三个整数x; Ÿ; z，其中x> 0且y> 0是由街道号z连接的交叉点的数量。 块的末尾由包含x = y = 0的行标记。在输入文件的末尾有一个空块，x = y = 0。

产量

每个块的输出一行包含街道编号序列（序列的单个成员由空格分隔），描述Johnny的往返行程。 如果找不到往返，则相应的输出块包含消息“往返不存在”。

#include<iostream>

#include<cstring>

#include<cstdio>

#define mst(a) memset(a,0,sizeof(a))

using namespace std;

const int maxn=20000+8;

int angleNum[maxn]; //表示顶点度数

int used[maxn]; //判断顶点是否走过

int n,k;

int res[maxn]; //路径数组

int cnt=0;

struct node{ //结构体定义 起点和终点

int a;

int b;

}rng[maxn];

bool is\_OK() //此方法用于判定每个顶点的度数

{

for(int i=1;i<=n;i++)

if(angleNum[i]%2) //如果存在一个顶点度数为偶数度，那么就不存在欧拉回路

return false; //仅仅对于有向图而言

return true;

}

void dfs(int x)

{

for(int i=1;i<=k;i++)

{

if(!used[i]&&(rng[i].a==x||rng[i].b==x)) //这里处理有点特殊，我们不确定

{ //走的是每个分支的起点还是终点

used[i]=1;

dfs(rng[i].a+rng[i].b-x);

res[++cnt]=i;

}

}

}

int main()

{

ios::sync\_with\_stdio(false); //cin提速

cin.tie(0);

int x,y,z;

while(cin>>x>>y&&(x+y))

{

int point=min(x,y); //求出我们的源点

n=max(x,y);

mst(angleNum);

mst(used);

cnt=k=0;

do{

cin>>z;

rng[z].a=x;

rng[z].b=y;

++k;

angleNum[x]++; angleNum[y]++;

n=max(n,max(x,y)); //求出最大的顶点数

}while(cin>>x>>y&&(x+y));

if(!is\_OK())

{

cout<<"Round trip does not exist."<<endl;

continue;

}

dfs(point); //dfs深搜欧拉回路

for(int i=cnt;i>=1;i--) //逆序输出路径

{

if(i!=1)

cout<<res[i]<<" ";

else

cout<<res[i]<<endl;

}

}

return 0;

}

### 容斥原理

Z城市居住着很多只跳蚤。在Z城市周六生活频道有一个娱乐节目。一只跳蚤将被请上一个高空钢丝的正中央。钢丝很长，可以看作是无限长。节目主持人会给该跳蚤发一张卡片。卡片上写有N+1个自然数。其中最后一个是M，而前N个数都不超过M，卡片上允许有相同的数字。跳蚤每次可以从卡片上任意选择一个自然数S，然后向左，或向右跳S个单位长度。而他最终的任务是跳到距离他左边一个单位长度的地方，并捡起位于那里的礼物。

比如当N=2，M=18时，持有卡片(10, 15, 18)的跳蚤，就可以完成任务：他可以先向左跳10个单位长度，然后再连向左跳3次，每次15个单位长度，最后再向右连跳3次，每次18个单位长度。而持有卡片(12, 15, 18)的跳蚤，则怎么也不可能跳到距他左边一个单位长度的地方。

当确定N和M后，显然一共有M^N张不同的卡片。现在的问题是，在这所有的卡片中，有多少张可以完成任务。

In 2 4 out 12

先将m质因数分解，然后容斥统计即可

#include<iostream>

#include<cmath>

using namespace std;

const int maxn=100+8;

typedef long long LL;

int num;

int prime[maxn];

LL p[maxn];

LL res,temp;

int n,m;

void divide(int m)

{

num=0;

for(int i=2;i\*i<=m;i++)

{

if(m%i==0)

{

prime[++num]=i;

m/=i;

while(m%i==0) m/=i;

}

}

if(m>1) prime[++num]=m;

}

void dfs(int b,int cnt,int c)

{

if(cnt==c)

{

int x=m;

for(int i=1;i<=c;i++)

x/=p[i];

temp+=pow(x,n);

return;

}

for(int i=b;i<=num;i++)

{

p[cnt+1]=prime[i];

dfs(i+1,cnt+1,c);

}

}

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

while(cin>>n>>m)

{

res=0;

divide(m);

for(int i=1;i<=num;i++)

{

temp=0;

dfs(1,0,i);

if(i&1) res+=temp;

else res-=temp;

}

res=pow(m,n)-res;

cout<<res<<endl;

}

return 0;

}

### 最大三角形面积（凸包+旋转卡壳）

老师给我们平面上n个点，让我们求出组合的三角形的最大面积

我们可以先将这n个点组成一个凸包，然后通过旋转卡壳求出凸包的最大直径

找出直径对应的两个点，然后通过遍历求出第三个点

这里面用上了叉积的概念，即一条边X另一条边

公式：

三角的面积等于同一个起点出发的两边的叉积\*1/2 假设A B是两条边

面积就等于（A+B）/2

3 3 4 2 6 3 7 1.50  
6 2 6 3 9 2 0 8 0 6 6 7 7 27.00

#include<iostream>

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<string>

#define mst(a) memset(a,0,sizeof(a))

using namespace std;

const int maxn=50000+8;

struct node{

int x;

int y;

}p[maxn],ch[maxn];

int n;

bool cmp(node a,node b)

{

if(a.x==b.x)

{

return a.y<b.y;

}

return a.x<b.x;

}

double Cross(node s,node a,node b)

{

int x1=a.x-s.x;

int y1=a.y-s.y;

int x2=b.x-s.x;

int y2=b.y-s.y;

return x1\*y2-x2\*y1;

}

int Andrew()

{

sort(p,p+n,cmp);

int m=0;

for(int i=0;i<n;i++)

{

//下

while(m>1&&Cross(ch[m-2],ch[m-1],p[i])<0) m--;

ch[m++]=p[i];

}

int k=m;

for(int i=n-2;i>=0;i--)

{

//上

while(m>k&&Cross(ch[m-2],ch[m-1],p[i])<0) m--;

ch[m++]=p[i];

}

if(n>1) m--;

return m;

}

int main()

{

while(cin>>n)

{

for(int i=0;i<n;i++)

{

cin>>p[i].x>>p[i].y;

}

int m=Andrew();

double res=0;

for(int i=0;i<m;i++)

{

int q=1;

for(int j=i+1;j<m;j++)

{

while(Cross(ch[i],ch[j],ch[q+1])>Cross(ch[i],ch[j],ch[q]))

q=(q+1)%m;

res=max(res,Cross(ch[i],ch[j],ch[q]));

}

}

printf("%.2lf\n",res/2.0);

}

return 0;

}

### 大数基本操作（Java）

import java.util.Scanner;

import java.math.BigInteger;

public class Main {

public static void main(String[] args) {

BigInteger[] a = new BigInteger[10100];

a[0] = BigInteger.valueOf(1);

for(int i = 1;i < 10100;i++) {

a[i] = a[i-1].multiply(BigInteger.valueOf(i));

}

Scanner in = new Scanner(System.in);

while (in.hasNextInt()) {

int x = in.nextInt();

System.out.println(a[x]);

}

}

}

### 大数加法（C++）

string add1(string s1, string s2)

{

if (s1 == "" && s2 == "") return "0";

if (s1 == "") return s2;

if (s2 == "") return s1;

string maxx = s1, minn = s2;

if (s1.length() < s2.length()){

maxx = s2;

minn = s1;

}

int a = maxx.length() - 1, b = minn.length() - 1;

for (int i = b; i >= 0; --i){

maxx[a--] += minn[i] - '0'; // a一直在减 ， 额外还要减个'0'

}

for (int i = maxx.length()-1; i > 0;--i){

if (maxx[i] > '9'){

maxx[i] -= 10;//注意这个是减10

maxx[i - 1]++;

}

}

if (maxx[0] > '9'){

maxx[0] -= 10;

maxx = '1' + maxx;

}

return maxx;

}

### 大数阶乘（C++）

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long LL;

const int maxn = 100010;

int num[maxn], len;

/\*

在mult函数中，形参部分：len每次调用函数都会发生改变，n表示每次要乘以的数，最终返回的是结果的长度

tip: 阶乘都是先求之前的(n-1)!来求n!

初始化Init函数很重要，不要落下

\*/

void Init() {

len = 1;

num[0] = 1;

}

int mult(int num[], int len, int n) {

LL tmp = 0;

for(LL i = 0; i < len; ++i) {

tmp = tmp + num[i] \* n; //从最低位开始，等号左边的tmp表示当前位，右边的tmp表示进位（之前进的位）

num[i] = tmp % 10; // 保存在对应的数组位置，即去掉进位后的一位数

tmp = tmp / 10; // 取整用于再次循环,与n和下一个位置的乘积相加

}

while(tmp) { // 之后的进位处理

num[len++] = tmp % 10;

tmp = tmp / 10;

}

return len;

}

int main() {

Init();

int n;

n = 1977; // 求的阶乘数

for(int i = 2; i <= n; ++i) {

len = mult(num, len, i);

}

for(int i = len - 1; i >= 0; --i)

printf("%d",num[i]); // 从最高位依次输出,数据比较多采用printf输出

printf("\n");

return 0;

}

# 常用函数与STL标准模板库

### STL实现Ugly Numbers

#include <iostream>

#include <queue>

/\*

\* Ugly Numbers

\* Ugly numbers are numbers whose only prime factors are 2, 3 or 5.

\* 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...

\*/

typedef std::pair<unsigned long, int> node\_type;

int main(int argc, const char \* argv[])

{

unsigned long result[1502];

std::priority\_queue<node\_type, std::vector<node\_type>, std::greater<node\_type>> Q;

Q.push(std::make\_pair(1, 2));

for (int i = 0; i < 1500; i++)

{

node\_type node = Q.top();

Q.pop();

switch (node.second)

{

case 2:

Q.push(std::make\_pair(node.first \* 2, 2));

case 3:

Q.push(std::make\_pair(node.first \* 3, 3));

case 5:

Q.push(std::make\_pair(node.first \* 5, 5));

}

result[i] = node.first;

}

int n;

std::cin >> n;

while (n > 0)

{

std::cout << result[n - 1] << '\n';

std::cin >> n;

}

return 0;

}

### STL-pair

STL的<utility>头文件中描述了一个看上去非常简单的模版类pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较运算符模版函数。   
Example，想要定义一个对象表示一个平面坐标点，则可以：

pair<double, double> p;

cin >> p.first >> p.second;

### STL-set的基本操作

s.begin() // 返回指向第一个元素的迭代器

s.clear() // 清除所有元素

s.count() // 返回某个值元素的个数

s.empty() // 如果集合为空，返回true(真）

s.end() // 返回指向最后一个元素之后的迭代器，不是最后一个元素

s.equal\_range() // 返回集合中与给定值相等的上下限的两个迭代器

s.erase() // 删除集合中的元素

s.find() // 返回一个指向被查找到元素的迭代器

s.get\_allocator() // 返回集合的分配器

s.insert() // 在集合中插入元素

s.lower\_bound() // 返回指向大于（或等于）某值的第一个元素的迭代器

s.key\_comp() // 返回一个用于元素间值比较的函数

s.max\_size() // 返回集合能容纳的元素的最大限值

s.rbegin() // 返回指向集合中最后一个元素的反向迭代器

s.rend() // 返回指向集合中第一个元素的反向迭代器

s.size() // 集合中元素的数目

s.swap() // 交换两个集合变量

s.upper\_bound() // 返回大于某个值元素的迭代器

s.value\_comp() // 返回一个用于比较元素间的值的函数

多重集合与集合的区别在于集合中不能存在相同元素，而多重集合中可以存在。

multiset<int> s;

multiset<double> ss;

multiset和set的基本操作相似，需要注意的是，集合的count()能返回0（无）或者1（有），而多重集合是有多少个返回多少个

### STL-vector

vector<int> s;

// 定义一个空的vector对象，存储的是int类型的元素

vector<int> s(n);

// 定义一个含有n个int元素的vector对象

vector<int> s(first, last);

// 定义一个vector对象，并从由迭代器first和last定义的序列[first, last)中复制初值

Vector的基本操作

s[i] // 直接以下标方式访问容器中的元素

s.front() // 返回首元素

s.back() // 返回尾元素

s.push\_back(x) // 向表尾插入元素x

s.size() // 返回表长

s.empty() // 表为空时，返回真，否则返回假

s.pop\_back() // 删除表尾元素

s.begin() // 返回指向首元素的随机存取迭代器

s.end() // 返回指向尾元素的下一个位置的随机存取迭代器

s.insert(it, val) // 向迭代器it指向的元素前插入新元素val

s.insert(it, n, val)// 向迭代器it指向的元素前插入n个新元素val

s.insert(it, first, last)

// 将由迭代器first和last所指定的序列[first, last)插入到迭代器it指向的元素前面

s.erase(it) // 删除由迭代器it所指向的元素

s.erase(first, last)// 删除由迭代器first和last所指定的序列[first, last)

s.reserve(n) // 预分配缓冲空间，使存储空间至少可容纳n个元素

s.resize(n) // 改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），元素默认值将填满扩展出的空间

s.resize(n, val) // 改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），val将填满扩展出的空间

s.clear() // 删除容器中的所有元素

s.swap(v) // 将s与另一个vector对象进行交换

s.assign(first, last)

// 将序列替换成由迭代器first和last所指定的序列[first, last)，[first, last)不能是原序列中的一部分

// 要注意的是，resize操作和clear操作都是对表的有效元素进行的操作，但并不一定会改变缓冲空间的大小

// 另外，vector还有其他的一些操作，如反转、取反等，不再一一列举

// vector上还定义了序列之间的比较操作运算符（>、<、>=、<=、==、!=），可以按照字典序比较两个序列。

### STL-stack

Stack的基本操作：

s.push(x); // 入栈

s.pop(); // 出栈

s.top(); // 访问栈顶

s.empty(); // 当栈空时，返回true

s.size(); // 访问栈中元素个数

### STL-queue

Queue的基本操作：

q.push(x); // 入队列

q.pop(); // 出队列

q.front(); // 访问队首元素

q.back(); // 访问队尾元素

q.empty(); // 判断队列是否为空

q.size(); // 访问队列中的元素个数

优先队列：

priority\_queue<int> q;

priority\_queue<pair<int, int> > qq; // 注意在两个尖括号之间一定要留空格，防止误判

priority\_queue<int, vector<int>, greater<int> > qqq;// 定义小的先出队列

优先队列的基本操作：

q.empty() // 如果队列为空，则返回true，否则返回false

q.size() // 返回队列中元素的个数

q.pop() // 删除队首元素，但不返回其值

q.top() // 返回具有最高优先级的元素值，但不删除该元素

q.push(item) // 在基于优先级的适当位置插入新元素

#include <iostream>

#include <queue>

using namespace std;

class T

{

public:

int x, y, z;

T(int a, int b, int c) : x(a), y(b), z(c) {}

};

bool operator < (const T &tOne, const T &tTwo)

{

return tOne.z < tTwo.z; // 按照z的顺序来决定tOne和tTwo的顺序

}

int main()

{

priority\_queue<T> q;

q.push(T(4, 4, 3));

q.push(T(2, 2, 5));

q.push(T(1, 5, 4));

q.push(T(3, 3, 6));

while (!q.empty())

{

T t = q.top();

q.pop();

cout << t.x << " " << t.y << " " << t.z << '\n';

}

return 0;

}

### STL-map

/\* 向map中插入元素 \*/

m[key] = value; // [key]操作是map很有特色的操作,如果在map中存在键值为key的元素对, 则返回该元素对的值域部分,否则将会创建一个键值为key的元素对,值域为默认值。所以可以用该操作向map中插入元素对或修改已经存在的元素对的值域部分。

m.insert(make\_pair(key, value)); // 也可以直接调用insert方法插入元素对,insert操作会返回一个pair,当map中没有与key相匹配的键值时,其first是指向插入元素对的迭代器,其second为true;若map中已经存在与key相等的键值时,其first是指向该元素对的迭代器,second为false。

/\* 查找元素 \*/

int i = m[key]; // 要注意的是,当与该键值相匹配的元素对不存在时,会创建键值为key（当另一个元素是整形时，m[key]=0）的元素对。

map<string, int>::iterator it = m.find(key); // 如果map中存在与key相匹配的键值时,find操作将返回指向该元素对的迭代器,否则,返回的迭代器等于map的end()(参见vector中提到的begin()和end()操作)。

/\* 删除元素 \*/

m.erase(key); // 删除与指定key键值相匹配的元素对,并返回被删除的元素的个数。

m.erase(it); // 删除由迭代器it所指定的元素对,并返回指向下一个元素对的迭代器。

/\* 其他操作 \*/

m.size(); // 返回元素个数

m.empty(); // 判断是否为空

m.clear(); // 清空所有元素

**二维map应用：**

map<string,map<string,int> > mp;

map<string,map<string,int> >::iterator it;

map<string,int>::iterator it2;

for(it=mp.begin();it!=mp.end();it++){

cout<<it->first<<endl;

for(it2=it->second.begin();it2!=it->second.end();it2++){

cout<<" |----"<<it2->first<<"("<<it2->second<<")"<<endl;

}

}

### STL-bitset

在 STLSTL 的头文件中 <bitset><bitset> 中定义了模版类 bitsetbitset，用来方便地管理一系列的 bitbit 位的类。bitsetbitset 除了可以访问指定下标的 bitbit 位以外，还可以把它们作为一个整数来进行某些统计。

bitsetbitset 模板类需要一个模版参数，用来明确指定含有多少位

定义bitset对象的示例代码：

const int MAXN = 32;

bitset<MAXN> bt; // bt 包括 MAXN 位，下标 0 ~ MAXN - 1，默认初始化为 0

bitset<MAXN> bt1(0xf); // 0xf 表示十六进制数 f，对应二进制 1111，将 bt1 低 4 位初始化为 1

bitset<MAXN> bt2(012); // 012 表示八进制数 12，对应二进制 1010，即将 bt2 低 4 位初始化为 1010

bitset<MAXN> bt3("1010"); // 将 bt3 低 4 位初始化为 1010

bitset<MAXN> bt4(s, pos, n);// 将 01 字符串 s 的 pos 位开始的 n 位初始化 bt4

bitset的基本操作：

bt.any() // bt 中是否存在置为 1 的二进制位？

bt.none() // bt 中不存在置为 1 的二进制位吗？

bt.count() // bt 中置为 1 的二进制位的个数

bt.size() // bt 中二进制位的个数

bt[pos] // 访问 bt 中在 pos 处的二进制位

bt.test(pos) // bt 中在 pos 处的二进制位是否为 1

bt.set() // 把 bt 中所有二进制位都置为 1

bt.set(pos) // 把 bt 中在 pos 处的二进制位置为 1

bt.reset() // 把 bt 中所有二进制位都置为 0

bt.reset(pos) // 把 bt 中在pos处的二进制位置为0

bt.flip() // 把 bt 中所有二进制位逐位取反

bt.flip(pos) // 把 bt 中在 pos 处的二进制位取反

bt[pos].flip() // 同上

bt.to\_ulong() // 用 bt 中同样的二进制位返回一个 unsigned long 值

os << bt // 把 bt 中的位集输出到 os 流

### STL-iterator迭代器特别输出

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> s;

s.push\_back(1);

s.push\_back(2);

s.push\_back(3);

copy(s.begin(), s.end(), ostream\_iterator<int> (cout, " "));

cout << '\n';

return 0;

}

这段代码中的copy就是STL中定义的一个模版函数，copy(s.begin(), s.end(), ostream\_iterator<int>(cout, ” “));的意思是将由s.begin()至s.end()（不含s.end()）所指定的序列复制到标准输出流out中，用” “作为每个元素的间隔。也就是说，这句话的作用其实就是将表中的所有内容依次输出

### STL-algorithm

min\_element/max\_element找出容器中的最小/最大值

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

vector<int> L;

for (int i=0; i<10; i++)

{

L.push\_back(i);

}

vector<int>::iterator min\_it = min\_element(L.begin(), L.end());

vector<int>::iterator max\_it = max\_element(L.begin(), L.end());

cout << "Min is " << \*min\_it << endl;

cout << "Max is " << \*max\_it << endl;

return 0;

}

Copy:

// 将vOne的前三个元素复制到vTwo的中间（覆盖掉原来数据）

copy(vOne.begin(), vOne.begin() + 3, vTwo.begin() + 4);

// 在vTwo内部进行复制，注意参数2表示结束位置，结束位置不参与复制

copy(vTwo.begin() + 4, vTwo.begin() + 7, vTwo.begin() + 2);

### 输出一个数的2 8 10 16进制（模板）

#include <bitset>

#include <iostream>

using namespace std;

int main()

{

cout << "36的8进制:" << std::oct << 36 << endl;

cout << "36的10进制" << std::dec << 36 << endl;

cout << "36的16进制:" << std::hex << 36 << endl;

cout << "36的2进制: " << bitset<8>(36) << endl;

return 0;

}

### 10进制与26进制相互转化

#include<iostream>

#include <algorithm>

using namespace std;

int main(){

long long ans=0;

string s;

cin>>s;

int len=s.size();

for(int i=0,j=1;i<s.size();i++,j\*=26){

ans+=(int)(s[len-i-1]-65)\*j;

}

cout<<ans<<endl; //26进制转十进制

string str="";

//ans=817;

while(ans>0){

int m=ans%26;

if(m==0) m=0;

str+=(char)(m+65);

ans=(ans-m)/26;

}

reverse(str.begin(),str.end()); //反序

cout<<str<<endl; // 十进制转26进制

return 0;

}

### 26进制下的大数加法

AAAADH BCE DRW UHD D AAAAA  
BFL XYZ D

#include<iostream>

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<string>

#define mst(a) memset(a,0,sizeof(a))

using namespace std;

const int maxn=200+10;

char s1[maxn],s2[maxn];

int a[maxn],b[maxn];

char s[maxn];

int main()

{

while(~scanf("%s %s",s1,s2))

{

mst(a);

mst(b);

int len1=strlen(s1);

int len2=strlen(s2);

for(int i=0;i<len1;i++)

{

int x=s1[i]-'A';

a[len1-i-1]=x;

}

for(int i=0;i<len2;i++)

{

int x=s2[i]-'A';

b[len2-i-1]=x;

}

int len=max(len1,len2);

int pre=0;

int k=0;

string res="";

for(int i=0;i<=len;i++)

{

int v=a[i]+b[i]+pre;

pre=0;

if(v>=26)

{

v-=26;

pre=1;

}

if(v!=0)

{

k=i;

}

s[i]=v;

}

for(int i=0;i<=k;i++)

{

res+=s[i]+'A';

}

reverse(res.begin(),res.end());

int n=res.length();

int flag=0;

for(int i=0;i<n;i++)

{

if(res[i]=='A')

{

flag=0;

}

else

{

flag=1;

break;

}

}

if(flag)

cout<<res<<endl;

else

cout<<'A'<<endl;

}

}

# 数论模板

### 快速幂模板

long long quick\_pow(long long x,long long num)

{

long long res=1;

while(num>0)

{

if(num&1)

{

res=(res\*x)%mod;

}

x=(x\*x)%mod;

num/=2;

}

return res;

}

### 矩阵快速幂

#include<bits/stdc++.h>

#define ll long long

#define mod(x) ((x)%MOD)

using namespace std;

const ll MOD = 1e9 + 7;

struct mat{

ll m[3][3];

}a,ans,unit;

void init() {

memset(unit.m,0,sizeof(unit.m));

memset(a.m,0,sizeof(a.m));

unit.m[0][0] = 1;

unit.m[1][1] = 1;

a.m[0][0] = 3;

a.m[0][1] = 1;

a.m[1][0] = 1;

a.m[1][1] = 3;

}

mat operator \* (mat m1,mat m2) {

mat t;

ll r;

for(int i = 0;i < 3;i++) {

for(int j = 0;j < 3;j++) {

r = 0;

for(int k = 0;k < 3;k++) {

r = mod(r\*1ll + mod(mod(m1.m[i][k])\*1ll\*mod(m2.m[k][j])));

}

t.m[i][j] = r;

}

}

return t;

}

mat quick\_pow(ll x) {

mat t = unit;

while(x) {

if(x & 1) {

t = t\*a;

}

a = a\*a;

x >>= 1;

}

return t;

}

int main(){

init();

ans = quick\_pow(n);

}

### 欧拉函数PHI

##### 分解质因数法

/\*

\* 分解质因数法求解，getFactor(n)函数见《合数相关》

\*/

int main(int argc, const char \* argv[])

{

// ...

getFactors(n);

int ret = n;

for (int i = 0; i < fatCnt; i++)

{

ret = (int)(ret / factor[i][0] \* (factor[i][0] - 1));

}

return 0;

}

##### 筛法欧拉函数

const int MAXN = 100;

int phi[MAXN + 2];

int main(int argc, const char \* argv[])

{

for (int i = 1; i <= MAXN; i++)

{

phi[i] = i;

}

for (int i = 2; i <= MAXN; i += 2)

{

phi[i] /= 2;

}

for (int i = 3; i <= MAXN; i += 2)

{

if (phi[i] == i)

{

for (int j = i; j <= MAXN; j += i)

{

phi[j] = phi[j] / i \* (i - 1);

}

}

}

return 0;

}

##### 单独求解

/\*

\* 单独求解的本质是公式的应用

\*/

unsigned euler(unsigned x)

{

unsigned i, res = x; // unsigned == unsigned int

for (i = 2; i < (int)sqrt(x \* 1.0) + 1; i++)

{

if (!(x % i))

{

res = res / i \* (i - 1);

while (!(x % i))

{

x /= i; // 保证i一定是素数

}

}

}

if (x > 1)

{

res = res / x \* (x - 1);

}

return res;

}

##### 线性筛

/\*

\* 同时得到欧拉函数和素数表

\*/

const int MAXN = 10000000;

bool check[MAXN + 10];

int phi[MAXN + 10];

int prime[MAXN + 10];

int tot; // 素数个数

void phi\_and\_prime\_table(int N)

{

memset(check, false, sizeof(check));

phi[1] = 1;

tot = 0;

for (int i = 2; i <= N; i++)

{

if (!check[i])

{

prime[tot++] = i;

phi[i] = i - 1;

}

for (int j = 0; j < tot; j++)

{

if (i \* prime[j] > N)

{

break;

}

check[i \* prime[j]] = true;

if (i % prime[j] == 0)

{

phi[i \* prime[j]] = phi[i] \* prime[j];

break;

}

else

{

phi[i \* prime[j]] = phi[i] \* (prime[j] - 1);

}

}

}

return ;

}

### 凸包模板

#include <iostream>

#include <algorithm>

#include <iomanip>

#include<stdio.h>

using namespace std;

const int maxn = 50010;

struct Point {

int x , y;

bool operator < (Point const &rhs) const {

return (x < rhs.x) || (x == rhs.x && y < rhs.y);

}

};

int Cross(Point const &O , Point const &A , Point const &B)

{

int xoa = A.x - O.x;

int xob = B.x - O.x;

int yoa = A.y - O.y;

int yob = B.y - O.y;

return xoa \* yob - xob \* yoa;

}

int Andrew(Point \*p , int n , Point \*ch)

{

sort(p , p + n);

int m = 0;

for(int i = 0; i < n; i++)

{ //下凸包

while(m > 1 && Cross(ch[m-2] , ch[m-1] , p[i]) < 0) m--;

ch[m++] = p[i];

}

int k = m;

for(int i = n - 2; i >= 0; i--) { //上凸包

while(m > k && Cross(ch[m-2] , ch[m-1] , p[i]) < 0) m--;

ch[m++] = p[i];

}

if(n > 1) m--;

return m;

}

Point p[maxn] , ch[maxn];

int main()

{

int n;

while(cin >> n)

{

for(int i = 0; i < n; i++) cin >> p[i].x >> p[i].y;

int m = Andrew(p , n , ch); ///求凸包

///旋转卡壳法

int ans = 0;

for(int i = 0; i < m; i++)

{

int q = 1;

for(int j = i + 1; j < m; j++)

{

while(Cross(ch[i],ch[j],ch[q+1]) > Cross(ch[i],ch[j],ch[q]))

q = (q + 1) % m;

ans = max(ans , Cross(ch[i],ch[j],ch[q]));

}

}

//cout << fixed << setprecision(2) << ans / 2.0 << endl;

printf("%.2lf\n",ans/2.0);

}

return 0;

}

### 欧几里得拓展GCD

扩展欧几里德算法是用来在已知a, b求解一组x，y，使它们满足贝祖等式： ax+by = gcd(a, b) =d（解一定存在，根据数论中的相关定理）。扩展欧几里德常用在求解模线性方程及方程组中。

ll gcd(ll a,ll b) {

return b == 0 ? a : gcd(b, a % b);

}

void exgcd(ll a, ll b, ll &d, ll &x, ll &y) {

if(!b) d=a,x=1,y=0;

else exgcd(b, a % b, d, y, x),y -= x \* (a / b);

}

### 线性方程组（高斯消元）

##### **列主元**

/\*

\* 列主元gauss消去求解a[][] \* x[] = b[]

\* 返回是否有唯一解，若有解在b[]中

\*/

#define fabs(x) ((x) > 0 ? (x) : (-x))

#define eps 1e-10

const int MAXN = 100;

int gaussCpivot(int n, double a[][MAXN], double b[])

{

int i, j, k, row = 0;

double MAXP, temp;

for (k = 0; k < n; k++)

{

for (MAXP = 0, i = k; i < n; i++)

{

if (fabs(a[i][k]) > fabs(MAXP))

{

MAXP = a[row = i][k];

}

}

if (fabs(MAXP) < eps)

{

return 0;

}

if (row != k)

{

for (j = k; j < n; j++)

{

temp = a[k][j];

a[k][j] = a[row][j];

a[row][j] = temp;

temp = b[k];

b[k] = b[row];

b[row] = temp;

}

}

for (j = k + 1; j < n; j++)

{

a[k][j] /= MAXP;

for (i = k + 1; i < n; i++)

{

a[i][j] -= a[i][k] \* a[k][j];

}

}

b[k] /= MAXP;

for (i = n - 1; i >= 0; i--)

{

for (j = i + 1; j < n; j++)

{

b[i] -= a[i][j] \* b[j];

}

}

}

return 1;

}

##### 全主元

/\*

\* 全主元gauss消去解a[][] \* x[] = b[]

\* 返回是否有唯一解，若有解在b[]中

\*/

#define fabs(x) ((x) > 0 ? (x) : (-x))

#define eps 1e-10

const int MAXN = 100;

int gaussTpivot(int n, double a[][MAXN], double b[])

{

int i, j, k, row = 0, col = 0, index[MAXN];

double MAXP, temp;

for (i = 0; i < n; i++)

{

index[i] = i;

}

for (k = 0; k < n; k++)

{

for (MAXP = 0, i = k; i < n; i++)

{

for (j = k; j < n; j++)

{

if (fabs(a[i][j] > fabs(MAXP)))

{

MAXP = a[row = i][col = j];

}

}

}

if (fabs(MAXP) < eps)

{

return 0;

}

if (col != k)

{

for (i = 0; i < n; i++)

{

temp = a[i][col];

a[i][col] = a[i][k];

a[i][k] = temp;

}

j = index[col];

index[col] = index[k];

index[k] = j;

}

if (row != k)

{

for (j = k; j < n; j++)

{

temp = a[k][j];

a[k][j] = a[row][j];

a[row][j] = temp;

}

temp = b[k];

b[k] = b[row];

b[row] = temp;

}

for (j = k + 1; j < n; j++)

{

a[k][j] /= MAXP;

for (i = k + 1; i < n; i++)

{

a[i][j] -= a[i][k] \* a[k][j];

}

}

b[k] /= MAXP;

for (i = k + 1; i < n; i++)

{

b[i] -= b[k] \* a[i][k];

}

}

for (i = n - 1; i >= 0; i--)

{

for (j = i + 1; j < n; j++)

{

b[i] -= a[i][j] \* b[j];

}

}

for (k = 0; k < n; k++)

{

a[0][index[k]] = b[k];

}

for (k = 0; k < n; k++)

{

b[k] = a[0][k];

}

return 1;

}

##### 高斯消元（自由变元，一类开关问题，位运算操作）

// 高斯消元法求方程组的解

const int MAXN = 300;

// 有equ个方程，var个变元。增广矩阵行数为equ，列数为var＋1，分别为0到var

int equ, var;

int a[MAXN][MAXN]; // 增广矩阵

int x[MAXN]; // 解集

int free\_x[MAXN]; // 用来存储自由变元（多解枚举自由变元可以使用）

int free\_num; // 自由变元的个数

// 返回值为－1表示无解，为0是唯一解，否则返回自由变元个数

int Gauss()

{

int max\_r, col, k;

free\_num = 0;

for (k = 0, col = 0; k < equ && col < var; k++, col++)

{

max\_r = k;

for (int i = k + 1; i < equ; i++)

{

if (abs(a[i][col]) > abs(a[max\_r][col]))

{

max\_r = i;

}

}

if (a[max\_r][col] == 0)

{

k--;

free\_x[free\_num++] = col; // 这是自由变元

continue;

}

if (max\_r != k)

{

for (int j = col; j < var + 1; j++)

{

swap(a[k][j], a[max\_r][j]);

}

}

for (int i = k + 1; i < equ; i++)

{

if (a[i][col] != 0)

{

for (int j = col; j < var + 1; j++)

{

a[i][j] ^= a[k][j];

}

}

}

}

for (int i = k; i < equ; i++)

{

if (a[i][col] != 0)

{

return -1; // 无解

}

}

if (k < var)

{

return var - k; // 自由变元个数

}

// 唯一解，回代

for (int i = var - 1; i >= 0; i--)

{

x[i] = a[i][var];

for (int j = i + 1; j < var; j++)

{

x[i] ^= (a[i][j] && x[j]);

}

}

return 0;

}

### 模线性方程（组）

##### 公共部分（拓展GCD）

int extgcd(int a, int b, int &x, int &y)

{

if (b == 0)

{

x = 1;

y = 0;

return a;

}

int d = extgcd(b, a % b, x, y);

int t = x;

x = y;

y = t - a / b \* y;

return d;

}

##### 模线性方程

/\*

\* 模线性方程 a \* x = b (% n)

\*/

void modeq(int a, int b, int n)

{

int e, i, d, x, y;

d = extgcd(b, a % b, x, y);

if (b % d > 0)

{

cout << "No answer!\n";

}

else

{

e = (x \* (b / d)) % n;

for (i = 0; i < d; i++)

{

cout << i + 1 << "-th ans:" << (e + i \* (n / d)) % n << '\n';

}

}

return ;

}

##### 模线性方程组（互质）

/\*

\* 模线性方程组

\* a = B[1](% W[1]); a = B[2](% W[2]); ... a = B[k](% W[k]);

\* 其中W，B已知，W[i] > 0且W[i]与W[j]互质，求a（中国剩余定理）

\*/

int china(int b[], int w[], int k)

{

int i, d, x, y, m, a = 0, n = 1;

for (i = 0; i < k; i++)

{

n \*= w[i]; // 注意不能overflow

}

for (i = 0; i < k; i++)

{

m = n / w[i];

d = extgcd(w[i], m, x, y);

a = (a + y \* m \* b[i]) % n;

}

if (a > 0)

{

return a;

}

else

{

return (a + n);

}

}

##### 模线性方程组（不要求互质）

typedef long long ll;

const int MAXN = 11;

int n, m;

int a[MAXN], b[MAXN];

int main(int argc, const char \* argv[])

{

int T;

cin >> T;

while (T--)

{

cin >> n >> m;

for (int i = 0; i < m; i++)

{

cin >> a[i];

}

for (int i = 0; i < m; i++)

{

cin >> b[i];

}

ll ax = a[0], bx = b[0], x, y;

int flag = 0;

for (int i = 1; i < m; i++)

{

ll d = extgcd(ax, a[i], x, y);

if ((b[i] - bx) % d != 0)

{

flag = 1; // 无整数解

break;

}

ll tmp = a[i] / d;

x = x \* (b[i] - bx) / d; // 约分

x = (x % tmp + tmp) % tmp;

bx = bx + ax \* x;

ax = ax \* tmp; // ax = ax \* a[i] / d

}

if (flag == 1 || n < bx)

{

puts("0");

}

else

{

ll ans = (n - bx) / ax + 1;

if (bx == 0)

{

ans--;

}

printf("%lld\n", ans);

}

}

return 0;

}

### 素数相关

##### 判断小于maxn的数是不是素数

/\*

\* 素数筛选，判断小于MAXN的数是不是素数

\* notprime是一张表，false表示是素数，true表示不是

\*/

const int MAXN = 1000010;

bool notprime[MAXN];

void init()

{

memset(notprime, false, sizeof(notprime));

notprime[0] = notprime[1] = true;

for (int i = 2; i < MAXN; i++)

{

if (!notprime[i])

{

if (i > MAXN / i) // 阻止后边i \* i溢出（或者i,j用long long)

{

continue;

}

// 直接从i \* i开始就可以，小于i倍的已经筛选过了

for (int j = i \* i; j < MAXN; j += i)

{

notprime[j] = true;

}

}

}

}

##### 查找出小于等于maxn的素数（生成连续素数表）

/\*

\* 素数筛选，查找出小于等于MAXN的素数

\* prime[0]存素数的个数

\*/

const int MAXN = 100000;

int prime[MAXN + 1];

void getPrime()

{

memset(prime, 0, sizeof(prime));

for (int i = 2; i <= MAXN; i++)

{

if (!prime[i])

{

prime[++prime[0]] = i;

}

for (int j = 1; j <= prime[0] && prime[j] <= MAXN / i; j++)

{

prime[prime[j] \* i] = 1;

if (i % prime[j] == 0)

{

break;

}

}

}

}

##### 随机素数测试

/\*

\* 随机素数测试（伪素数原理）

\* CALL: bool res = miller(n);

\* 快速测试n是否满足素数的“必要”条件，出错概率极低

\* 对于任意奇数n > 2和正整数s，算法出错概率≤2^(-s)

\*/

int witness(int a, int n)

{

int x, d = 1;

int i = ceil(log(n - 1.0) / log(2.0)) - 1;

for (; i >= 0; i--)

{

x = d;

d = (d \* d) % n;

if (d == 1 && x != 1 && x != n - 1)

{

return 1;

}

if (((n - 1) & (1 << i)) > 0)

{

d = (d \* a) % n;

}

}

return (d == 1 ? 0 : 1);

}

int miller(int n, int s = 50)

{

if (n == 2) // 质数返回1

{

return 1;

}

if (n % 2 == 0) // 偶数返回0

{

return 0;

}

int j, a;

for (j = 0; j < a; j++)

{

a = rand() \* (n - 2) / RAND\_MAX + 1;

// rand()只能随机产生[0, RAND\_MAX)内的整数

// 而且这个RAND\_MAX只有32768直接%n的话是永远

// 也产生不了[RAND\_MAX, n)之间的数

if (witness(a, n))

{

return 0;

}

}

return 1;

}

### 合数相关

##### 合数分解

/\*

\* 合数的分解需要先进行素数的筛选

\* factor[i][0]存放分解的素数

\* factor[i][1]存放对应素数出现的次数

\* fatCnt存放合数分解出的素数个数(相同的素数只算一次)

\*/

const int MAXN = 10000;

int prime[MAXN + 1];

// 获取素数

void getPrime()

{

memset(prime, 0, sizeof(prime));

for (int i = 2; i <= MAXN; i++)

{

if (!prime[i])

{

prime[++prime[0]] = i;

}

for (int j = 1; j <= prime[0] && prime[j] <= MAXN / i; j++)

{

prime[prime[j] \* i] = 1;

if (i % prime[j] == 0)

{

break;

}

}

}

return ;

}

long long factor[100][2];

int fatCnt;

// 合数分解

int getFactors(long long x)

{

fatCnt = 0;

long long tmp = x;

for (int i = 1; prime[i] <= tmp / prime[i]; i++)

{

factor[fatCnt][1] = 0;

if (tmp % prime[i] == 0)

{

factor[fatCnt][0] = prime[i];

while (tmp % prime[i] == 0)

{

factor[fatCnt][1]++;

tmp /= prime[i];

}

fatCnt++;

}

}

if (tmp != 1)

{

factor[fatCnt][0] = tmp;

factor[fatCnt++][1] = 1;

}

return fatCnt;

}

### 组合数学相关

定理

One

{1, 2, … n}的r组合a1, a2, … ar出现在所有r组合中的字典序位置编号, C(n, m)表示n中取m的组合数

index = C(n, r) - C(n - a1, r) - C(n - a2, r - 1) - … - C(n - ar, 1)

Two

k \* C(n, k) = n \* C(n - 1, k - 1);

C(n, 0) + C(n, 2) + … = C(n, 1) + C(n, 3) + …

1 \* C(n, 1) + 2 \* C(n, 2) + … + n \* C(n, n) = n \* 2^(n - 1)

Three · Catalan数

C\_n = C(2 \* n, n) / (n + 1)

C\_n = (4 \* n - 2) / (n + 1) \* C\_n - 1

C\_1 = 1

Four · Stirling数 · 1

s(p, k)是将p个物体排成k个非空的循环排列的方法数(或者: 把p个人排成k个非空圆圈的方法数)。

s(p, k) = (p - 1) \* s(p - 1, k) + s(p - 1, k - 1);

Five · Stirling数 · 2

S(p, k) = k \* S(p - 1, k) + S(p - 1, k - 1).

S(p, 0) = 0, (p >= 1);

S(p, p) = 1, (p >= 0);

且有 S(p, 1) = 1, (p >= 1);

S(p, 2) = 2^(p - 1) - 1, (p >= 2);

S(p, p - 1) = C(p, 2);

Six · Bell数

B\_p = S(p, 0) + S(p, 1) + … + S(p, p)

B\_p = C(p - 1, 0) \* B\_0 + C(p - 1, 1) \* B\_1 + … + C(p - 1, p - 1) \* B\_(p - 1)

##### 组合数C(n,r)

int com(int n, int r) // return C(n, r)

{

if (n - r > r)

{

r = n - r; // C(n, r) = C(n, n - r)

}

int i, j, s = 1;

for (i = 0, j = 1; i < r; ++i)

{

s \*= (n - i);

for (; j <= r && s % j == 0; ++j)

{

s /= j;

}

}

return s;

}

##### 组合数C(a,b)预处理

typedef long long ll;

const ll MOD = 1e9 + 7; // 必须为质数才管用

const ll MAXN = 1e5 + 3;

ll fac[MAXN]; // 阶乘

ll inv[MAXN]; // 阶乘的逆元

ll QPow(ll x, ll n)

{

ll ret = 1;

ll tmp = x % MOD;

while (n)

{

if (n & 1)

{

ret = (ret \* tmp) % MOD;

}

tmp = tmp \* tmp % MOD;

n >>= 1;

}

return ret;

}

void init()

{

fac[0] = 1;

for (int i = 1; i < MAXN; i++)

{

fac[i] = fac[i - 1] \* i % MOD;

}

inv[MAXN - 1] = QPow(fac[MAXN - 1], MOD - 2);

for (int i = MAXN - 2; i >= 0; i--)

{

inv[i] = inv[i + 1] \* (i + 1) % MOD;

}

}

ll C(ll a, ll b)

{

if (b > a)

{

return 0;

}

if (b == 0)

{

return 1;

}

return fac[a] \* inv[b] % MOD \* inv[a - b] % MOD;

}

##### 集合划分问题

/\*

\* n元集合分划为k类的方案数记为S(n, k),称为第二类Stirling数。

\* 如{A,B,C}可以划分{{A}, {B}, {C}}, {{A, B}, {C}}, {{B, C}, {A}}, {{A, C}, {B}}, {{A, B, C}}。

\* 即一个集合可以划分为不同集合(1...n个)的种类数

\* CALL: compute(N); 每当输入一个n,输出B[n]

\*/

const int N = 2001;

int data[N][N], B[N];

void NGetM(int m, int n) // m 个数 n 个集合

{

// data[i][j]: i个数分成j个集合

int min, i, j;

data[0][0] = 1;

for (i = 1; i <= m; i++)

{

data[i][0] = 0;

}

for (i = 0; i <= m; i++)

{

data[i][i + 1] = 0;

}

for (i = 1; i <= m; i++)

{

if (i < n)

{

min = i;

}

else

{

min = n;

}

for (j = 1; j <= min; j++)

{

data[i][j] = (j \* data[i - 1][j] + data[i - 1][j - 1]);

}

}

return ;

}

void compute(int m)

{

// b[i]: Bell数

NGetM(m, m);

memset(B, 0, sizeof(B));

int i, j;

for (i=1; i <= m; i++)

{

for (j = 0; j <= i; j++)

{

B[i] += data[i][j];

}

}

return ;

}

##### 卢卡斯定理（从（1,1）到（n,m）的走法，机器人走方格问题）

#define MOD 1000000007

typedef long long LL;

LL quickPower(LL a, LL b)

{

LL ans = 1;

a %= MOD;

while (b)

{

if (b & 1)

{

ans = ans \* a % MOD;

}

b >>= 1;

a = a \* a % MOD;

}

return ans;

}

LL c(LL n, LL m)

{

if (m > n)

{

return 0;

}

LL ans = 1;

for (int i = 1; i <= m; i++)

{

LL a = (n + i - m) % MOD;

LL b = i % MOD;

ans = ans \* (a \* quickPower(b, MOD - 2) % MOD) % MOD;

}

return ans;

}

LL lucas(LL n, LL m)

{

if (m == 0)

{

return 1;

}

return c(n % MOD, m % MOD) \* lucas(n / MOD, m / MOD) % MOD;

}

int main(int argc, const char \* argv[])

{

LL n, m;

while (~scanf("%lld %lld", &n, &m))

{

LL max, min;

max = n + m - 3;

min = m - 1;

printf("%lld\n", lucas(max - 1, min - 1));

}

return 0;

}

### Polya计数

/\*

\* c种颜色的珠子，组成长为s的项链，项链没有方向和起始位置

\*/

int gcd(int a, int b)

{

return b ? gcd(b, a % b) : a;

}

int main(int argc, const char \* argv[])

{

int c, s;

while (cin >> c >> s)

{

int k;

long long p[64];

p[0] = 1; // power of c

for (k = 0; k < s; k++)

{

p[k + 1] = p[k] \* c;

}

// reflection part

long long count = s & 1 ? s \* p[s / 2 + 1] : (s / 2) \* (p[s / 2] + p[s / 2 + 1]);

// rotation part

for (k = 1 ; k <= s ; k++)

{

count += p[gcd(k, s)];

count /= 2 \* s;

}

cout << count << '\n';

}

return 0;

}

### 最大1矩阵

（全是1的最大子矩阵）

const int N = 1000;

bool a[N][N];

int Run(const int &m, const int &n) // a[1...m][1...n]

{ // O(m\*n)

int i, j, k, l, r, max=0;

int col[N];

for (j = 1; j <= n; j++)

{

if (a[1][j] == 0 )

{

col[j] = 0;

}

else

{

for (k = 2; k <= m && a[k][j] == 1; k++);

col[j] = k - 1;

}

}

for (i = 1; i <= m; i++)

{

if (i > 1)

{

for (j = 1; j <= n; j++)

{

if (a[i][j] == 0)

{

col[j] = 0;

}

else

{

if (a[i - 1][j] == 0)

{

for (k = i + 1; k <= m && a[k][j] == 1; k++);

col[j] = k-1;

}

}

}

}

for (j = 1; j <= n; j++)

{

if (col[j] >= i)

{

for (l = j - 1; l > 0 && col[l] >= col[j]; --l);

l++;

for (r = j + 1; r <= n && col[r] >= col[j]; ++r);

r--;

int res = (r - l + 1) \* (col[j] - i + 1);

if( res > max )

{

max = res;

}

}

}

}

return max;

}

### 约瑟夫环问题

/\*

\* n个人(编号 1...n),先去掉第m个数,然后从m+1个开始报1,

\* 报到k的退出,剩下的人继续从1开始报数.求胜利者的编号.

\*/

int main(int argc, const char \* argv[])

{

int n, k, m;

while (cin >> n >> k >> m, n || k || m)

{

int i, d, s = 0;

for (i = 2; i <= n; i++)

{

s = (s + k) % i;

}

k = k % n;

if (k == 0)

{

k = n;

}

d = (s + 1) + (m - k);

if (d >= 1 && d <= n)

{

cout << d << '\n';

}

else if (d < 1)

{

cout << n + d << '\n';

}

else if (d > n)

{

cout << d % n << '\n';

}

}

return 0;

}

##### 函数图像法

/\*

\* n 个人数到 k 出列，最后剩下的人编号

\*/

unsigned long long n, k;

int main()

{

cin >> n >> k;

long long y = k % 2;

long long x = 2, t = 0;

long long z1 = y, z2 = x;

while (x <= n)

{

z1 = y;

z2 = x;

t = (x - y) / (k - 1);

if (t == 0)

{

t++;

}

y = y + t \* k - ((y + t \* k) / (x + t)) \* (x + t);

x += t;

}

cout << (z1 + (n - z2) \* k) % n + 1 << endl;

return 0;

}

### 博弈论

##### Bash

#define \_MAX 10000

int a[\_MAX];

int b[\_MAX];

int bash(int N, int K)

{

if (N % (K + 1) == 0)

{

return 2;

}

return 1;

}

int main()

{

int T;

scanf("%d", &T);

for (int i = 0; i < T; i++)

{

scanf("%d%d", a + i, b + i);

}

for (int i = 0; i < T; i++)

{

if (bash(a[i], b[i]) == 1)

{

printf("A\n");

}

else

{

printf("B\n");

}

}

return 0;

}

##### Nim

int main(int argc, const char \* argv[])

{

int N, stone, tag = 0;

scanf("%d", &N);

while (N--)

{

scanf("%d", &stone);

tag ^= stone;

}

//tag为0则为后手赢，否则为先手赢

printf("%c\n", tag == 0 ? 'B' : 'A');

return 0;

}

##### SG打表

const int MAX\_DIG = 64;

// SG打表

// f[]:可以取走的石子个数

// sg[]:0~n的SG函数值

// hash[]:mex{}

int f[MAX\_DIG];

int sg[MAX\_DIG];

int hash[MAX\_DIG];

void getSG(int n)

{

memset(sg, 0, sizeof(sg));

for (int i = 1; i <= n; i++)

{

memset(hash, 0, sizeof(hash));

for (int j = 1; f[j] <= i; j++)

{

hash[sg[i - f[j]]] = 1;

}

for (int j = 0; j <= n; j++) // 求mes{}中未出现的最小的非负整数

{

if (hash[j] == 0)

{

sg[i] = j;

break;

}

}

}

}

##### SG DFS

const int MAX\_DIG = 64;

// DFS

// 注意 S数组要按从小到大排序 SG函数要初始化为-1 对于每个集合只需初始化1遍

// n是集合s的大小 S[i]是定义的特殊取法规则的数组

int s[MAX\_DIG];

int sg[MAX\_DIG \* 100];

int n;

int SG\_dfs(int x)

{

if (sg[x] != -1)

{

return sg[x];

}

bool vis[MAX\_DIG];

memset(vis, 0, sizeof(vis));

for (int i = 0; i < n; i++)

{

if (x >= s[i])

{

SG\_dfs(x - s[i]);

vis[sg[x - s[i]]] = 1;

}

}

int e;

for (int i = 0; ; i++)

{

if (!vis[i])

{

e = i;

break;

}

}

return sg[x] = e;

}

##### Wythoff

int main()

{

int t, a, b, m, k;

scanf("%d", &t);

while (t--)

{

scanf("%d%d", &a, &b);

if (a > b)

{

a ^= b;

b ^= a;

a ^= b;

}

m = b - a;

k = (int)(m \* (1 + sqrt(5)) / 2.0);

//m = ? \* a

//k = m / ?

//?:黄金分割数

//如果a == k，则为后手赢，否则先手赢（奇异局）

printf("%s\n", a == k ? "B" : "A");

}

return 0;

}

### 周期性方程

##### 追赶法解周期性方程

/\*

\* 周期性方程定义(n = 5)

\* |a\_1 b\_1 c\_1 d\_1 e\_1| = x\_1 --- 1

\* |e\_2 a\_2 b\_2 c\_2 d\_2| = x\_2 --- 2

\* |d\_2 e\_2 a\_2 b\_2 c\_2| = x\_3 --- 3

\* |c\_4 d\_2 e\_2 a\_4 b\_4| = x\_4 --- 4

\* |b\_5 c\_5 d\_5 e\_5 a\_5| = x\_5 --- 5

\* 输入： a[], b[], c[], x[]

\* 输出： 求解结果x在x[]中

\*/

const int MAXN = 1000;

int a[MAXN];

int b[MAXN];

int c[MAXN];

int x[MAXN];

void run()

{

c[0] /= b[0];

a[0] /= b[0];

x[0] /= b[0];

for (int i = 1; i < MAXN - 1; i++)

{

double temp = b[i] - a[i] \* c[i - 1];

c[i] /= temp;

x[i] = (x[i] - a[i] \* x[i - 1]) / temp;

a[i] = -a[i] \* a[i - 1] / temp;

}

a[MAXN - 2] = -a[MAXN - 2] - c[MAXN - 2];

for (int i = MAXN - 3; i >= 0; i--)

{

a[i] = -a[i] - c[i] \* a[i + 1];

x[i] -= c[i] \* x[i + 1];

}

x[MAXN - 1] -= (c[MAXN - 1] \* x[0] + a[MAXN - 1] \* x[MAXN - 2]);

x[MAXN - 1] /= (c[MAXN - 1] \* a[0] + a[MAXN - 1] \* a[MAXN - 2] + b[MAXN - 1]);

for (int i = MAXN - 2; i >= 0; i --)

{

x[i] += a[i] \* x[MAXN - 1];

}

return ;

}

### 阶乘

##### 阶乘最后非零位

/\*

\* 阶乘最后非零位 复杂度O(nlongn)

\* 返回改为，n以字符串方式传入

\*/

#define MAXN 10000

const int mod[20] = {1, 1, 2, 6, 4, 2, 2, 4, 2, 8, 4, 4, 8, 4, 6, 8, 8, 6, 8, 2};

int lastDigit(char \*buf)

{

int len = (int)strlen(buf);

int a[MAXN], i, c, ret = 1;

if (len == 1)

{

return mod[buf[0] - '0'];

}

for (i = 0; i < len; i++)

{

a[i] = buf[len - 1 - i] - '0';

}

for (; len; len -= !a[len - 1])

{

ret = ret \* mod[a[1] % 2 \* 10 + a[0]] % 5;

for (c = 0, i = len - 1; i >= 0; i--)

{

c = c \* 10 + a[i];

a[i] = c / 5;

c %= 5;

}

}

return ret + ret % 2 \* 5;

}

##### N的阶乘的长度

#define PI 3.1415926

int main()

{

int n, a;

while (~scanf(“%d", &n))

{

a = (int)((0.5 \* log(2 \* PI \* n) + n \* log(n) - n) / log(10));

printf("%d\n", a + 1);

}

return 0;

}

### 排列组合

##### 类循环排列

用递归实现多重循环,本递归程序相当于n重循环,每重循环的长度为m的情况,所以输出共有m^n行。

/\*

\* 输入样例: 3 2

\* 输出样例:

\* 0 0 0

\* 0 0 1

\* 0 1 0

\* 0 1 1

\* 1 0 0

\* 1 0 1

\* 1 1 0

\* 1 1 1

\*/

#define MAX\_N 10

int n, m; // 相当于n重循环,每重循环长度为m

int rcd[MAX\_N]; // 记录每个位置填的数

void loop\_permutation(int l)

{

int i;

if (l == n) // 相当于进入了 n 重循环的最内层

{

for (i = 0; i < n; i++)

{

cout << rcd[i];

if (i < n-1)

{

cout << " ";

}

}

cout << "\n";

return ;

}

for (i = 0; i < m; i++) // 每重循环长度为m

{

rcd[l] = i; // 在l位置放i

loop\_permutation(l + 1); // 填下一个位置

}

}

int main()

{

while (cin >> n >> m)

{

loop\_permutation(0);

}

return 0;

}

##### 全排列

/\*

\* 对输入的n个数作全排列。

\* 输入样例:

\* 3

\* 1 2 3

\* 输出样例:

\* 123

\* 132

\* 213

\* 231

\* 312

\* 321

\*/

#define MAX\_N 10

int n; // 共n个数

int rcd[MAX\_N]; // 记录每个位置填的数

int used[MAX\_N]; // 标记数是否用过

int num[MAX\_N]; // 存放输入的n个数

void full\_permutation(int l)

{

int i;

if (l == n)

{

for (i = 0; i < n; i++)

{

printf("%d", rcd[i]);

if (i < n-1)

{

printf(" ");

}

}

printf("\n");

return ;

}

for (i = 0; i < n; i++) // 枚举所有的数(n个),循环从开始

if (!used[i])

{ // 若num[i]没有使用过, 则标记为已使用

used[i] = 1;

rcd[l] = num[i]; // 在l位置放上该数

full\_permutation(l+1); // 填下一个位置

used[i] = 0; // 清标记

}

}

int read\_data()

{

int i;

if (scanf("%d", &n) == EOF)

{

return 0;

}

for (i = 0; i < n; i++)

{

scanf("%d", &num[i]);

}

for (i = 0; i < n; i++)

{

used[i] = 0;

}

return 1;

}

int main()

{

while (read\_data())

{

full\_permutation(0);

}

return 0;

}

程序通过used数组,标记数是否被用过,可以产生全排列,共有n!种。但是, 通过观察会发现,若输入的n个数有重复,那么在输出的n!种排列中,必然存在重复的项

##### 不重复排列

/\*

\* 输入n个数,输出由这n个数构成的排列,不允许出现重复的项。

\* 输入样例:

\* 3

\* 1 1 2

\* 输出样例:

\* 1 1 2

\* 1 2 1

\* 2 1 1

\*/

#define MAX\_N 10

int n, m; // 共有n个数,其中互不相同的有m个

int rcd[MAX\_N]; // 记录每个位置填的数

int used[MAX\_N]; // 标记m个数可以使用的次数

int num[MAX\_N]; // 存放输入中互不相同的m个数

void unrepeat\_permutation(int l)

{

int i;

if (l == n) // 填完了n个数,则输出

{

for (i = 0; i < n; i++)

{

printf("%d", rcd[i]);

if (i < n - 1)

{

printf(" ");

}

}

printf("\n");

return ;

}

for (i = 0; i < m; i++) // 枚举m个本质不同的数

{

if (used[i] > 0) // 若数num[i]还没被用完,则可使用次数减

{

used[i]--;

rcd[l] = num[i]; // 在l位置放上该数

unrepeat\_permutation(l+1); // 填下一个位置

used[i]++; // 可使用次数恢复

}

}

}

int read\_data()

{

int i, j, val;

if (scanf("%d", &n) == EOF)

{

return 0;

}

m = 0;

for (i = 0; i < n; i++)

{

scanf("%d", &val);

for (j = 0; j < m; j++)

{

if (num[j] == val)

{

used[j]++; break;

}

}

if (j == m)

{

num[m] = val;

used[m++] = 1;

}

}

return 1;

}

int main()

{

while (read\_data())

{

unrepeat\_permutation(0);

}

return 0;

}

本程序将全排列中的used标记数组改为记录输入中每个本质不同的数出现的次数,并在递归函数中使用。需要注意的是,在输入过程中,应剔除重复的数值。实际上,重复排列的产生是由于同一个位置被多次填入了相同的数,并且这多次填入又是在同一次循环过程中完成的。本方法通过统计每个本质不同的数的个数,使得循环长度由n变为m,这样,i一旦自增,就再也不会指向原先填入过的数了。这种方法剪去了重复项的生成,从而加快了搜索速度,是深度优先搜索中常用的剪枝技巧。

##### 一般组合

/\*

\* 输入n个数,从中选出m个数可构成集合,输出所有这样的集合。

\* 输入样例:

\* 4 3

\* 1 2 3 4

\* 输出样例:

\* 1 2 3

\* 1 2 4

\* 1 3 4

\* 2 3 4

\*/

#define MAX\_N 10

int n, m; // 从n个数中选出m个构成组合

int rcd[MAX\_N]; // 记录每个位置填的数

int num[MAX\_N]; // 存放输入的n个数

void select\_combination(int l, int p)

{

int i;

if (l == m) // 若选出了m个数, 则打印

{

for (i = 0; i < m; i++)

{

printf("%d", rcd[i]);

if (i < m - 1)

{

printf(" ");

}

}

printf("\n");

return ;

}

for (i = p; i < n; i++) // 上个位置填的是num[p-1],本次从num[p]开始试探

{

rcd[l] = num[i]; // 在l位置放上该数

select\_combination(l + 1, i + 1); // 填下一个位置

}

}

int read\_data()

{

int i;

if (scanf("%d%d", &n, &m) == EOF)

{

return 0;

}

for (i = 0; i < n; i++)

{

scanf("%d", &num[i]);

}

return 1;

}

int main()

{

while (read\_data())

{

select\_combination(0, 0);

}

return 0;

}

因为在组合生成过程中引入了变量 p,保证了每次填入的数字在num中的下标是递增的,所以不需要使用used进行标记,共C(n, m)种组合。

##### 全组合

/\*

\* 输入n个数,求这n个数构成的集合的所有子集。

\* 输入样例:

\* 3

\* 1 2 3

\* 输出样例:

\* 1

\* 1 2

\* 1 2 3

\* 1 3

\* 2

\* 2 3

\* 3

\*/

#define MAX\_N 10

int n; // 共n个数

int rcd[MAX\_N]; // 记录每个位置填的数

int num[MAX\_N]; // 存放输入的n个数

void full\_combination(int l, int p)

{

int i;

for (i = 0; i < l; i++) // 每次进入递归函数都输出

{

printf("%d", rcd[i]);

if (i < l-1)

{

printf(" ");

}

}

printf("\n");

for (i = p; i < n; i++) // 循环同样从p开始,但结束条件变为i>=n

{

rcd[l] = num[i]; // 在l位置放上该数

full\_combination(l + 1, i + 1); // 填下一个位置

}

}

int read\_data()

{

int i;

if (scanf("%d", &n) == EOF)

{

return 0;

}

for (i = 0; i < n; i++)

{

scanf("%d", &num[i]);

}

return 1;

}

int main()

{

while (read\_data())

{

full\_combination(0, 0);

}

return 0;

}

全组合,共2^n种,包含空集和自身。与全排列一样,若输入的n个数有重复, 那么在输出的2^n种组合中,必然存在重复的项。避免重复的方法与不重复排列算法中使用的类似。

##### 不重复组合

/\*

\* 输入n个数,求这n个数构成的集合的所有子集,不允许输出重复的项。

\* 输入样例:

\* 3

\* 1 1 3

\* 输出样例:

\* 1

\* 1 1

\* 1 1 3

\* 1 3

\* 3

\*/

#define MAX\_N 10

int n, m; // 输入n个数,其中本质不同的有m个

int rcd[MAX\_N]; // 记录每个位置填的数

int used[MAX\_N]; // 标记m个数可以使用的次数

int num[MAX\_N]; // 存放输入中本质不同的m个数

void unrepeat\_combination(int l, int p)

{

int i;

for (i = 0; i < l; i++) // 每次都输出

{

printf("%d", rcd[i]);

if (i < l - 1)

{

printf(" ");

}

}

printf("\n");

for (i = p; i < m; i++) // 循环依旧从p开始,枚举剩下的本质不同的数

{

if (used[i] > 0) // 若还可以用, 则可用次数减

{

used[i]--;

rcd[l] = num[i]; // 在l位置放上该

unrepeat\_combination(l+1, i); // 填下一个位置

used[i]++; //可用次数恢复

}

}

}

int read\_data()

{

int i, j, val;

if (scanf("%d", &n) == EOF)

{

return 0;

}

m = 0;

for (i = 0; i < n; i++)

{

scanf("%d", &val);

for (j = 0; j < m; j++)

{

if (num[j] == val)

{

used[j]++;

break;

}

}

if (j == m)

{

num[m] = val;

used[m++] = 1;

}

}

return 1;

}

int main()

{

while (read\_data())

{

unrepeat\_combination(0, 0);

}

return 0;

}

需要注意的是递归调用时,第二个参数是i,不再是全组合中的i+1!

搜索问题中有很多本质上就是排列组合问题,只不过加上了某些剪枝和限制条件。解这类题目的基本算法框架常常是类循环排列、全排列、一般组合或全组 合。而不重复排列与不重复组合则是两种非常有效的剪枝技巧。

### 求逆元

方程ax≡1(mod p),的解称为a关于模p的逆，当gcd(a,p)==1（即a，p互质）时，方程有唯一解，否则无解。

对于一些题目会要求把结果MOD一个数，通常是一个较大的质数，对于加减乘法通过同余定理可以直接拆开计算，但对于(a/b)%MOD这个式子，是不可以写成(a%MOD/b%MOD)%MOD的，但是可以写为(a\*b^-1)%MOD,其中b^-1表示b的逆元。

ll getinv (ll a,ll p) {

ll d, x, y;

exgcd (a, p, d, x, y);

return (x + p) % p == 0 ? p : (x + p) % p;

}

##### 拓展欧几里得法

/\*

\* 扩展欧几里得法（求ax + by = gcd）

\*/

// 返回d = gcd(a, b);和对应于等式ax + by = d中的x、y

long long extendGcd(long long a, long long b, long long &x, long long &y)

{

if (a == 0 && b == 0)

{

return -1; // 无最大公约数

}

if (b == 0)

{

x = 1;

y = 0;

return a;

}

long long d = extendGcd(b, a % b, y, x);

y -= a / b \* x;

return d;

}

// 求逆元 ax = 1(mod n)

long long modReverse(long long a, long long n)

{

long long x, y;

long long d = extendGcd(a, n, x, y);

if (d == 1)

{

return (x % n + n) % n;

}

else

{

return -1; // 无逆元

}

}

##### 简洁写法

/\*

\* 简洁写法I

\* 只能求a < m的情况，且a与m互质

\* 求ax = 1(mod m)的x值，即逆元(0 < a < m)

\*/

long long inv(long long a, long long m)

{

if (a == 1)

{

return 1;

}

return inv(m % a, m) \* (m - m / a) % m;

}

##### 欧拉函数法

/\*

\* 欧拉函数法

\* a 和 m 互质

\*/

// 快速幂取模

long long powM(long long a, long long b, long long m)

{

long long tmp = 1;

if (b == 0)

{

return 1;

}

if (b == 1)

{

return a % m;

}

tmp = powM(a, b >> 1, m);

tmp = tmp \* tmp % m;

if (b & 1)

{

tmp = tmp \* a % m;

}

return tmp;

}

long long inv(long long a, long long m)

{

return powM(a, m - 2, m);

}

##### 欧拉函数法（求阶乘逆元）

typedef long long ll;

const ll MOD = 1e9 + 7; // 必须为质数才管用

const ll MAXN = 1e5 + 3;

ll fac[MAXN]; // 阶乘

ll inv[MAXN]; // 阶乘的逆元

ll QPow(ll x, ll n)

{

ll ret = 1;

ll tmp = x % MOD;

while (n)

{

if (n & 1)

{

ret = (ret \* tmp) % MOD;

}

tmp = tmp \* tmp % MOD;

n >>= 1;

}

return ret;

}

void init()

{

fac[0] = 1;

for (int i = 1; i < MAXN; i++)

{

fac[i] = fac[i - 1] \* i % MOD;

}

inv[MAXN - 1] = QPow(fac[MAXN - 1], MOD - 2);

for (int i = MAXN - 2; i >= 0; i--)

{

inv[i] = inv[i + 1] \* (i + 1) % MOD;

}

}

### FFT

const double PI = acos(-1.0);

// 复数结构体

struct Complex

{

double x, y; // 实部和虚部 x + yi

Complex(double \_x = 0.0, double \_y = 0.0)

{

x = \_x;

y = \_y;

}

Complex operator - (const Complex &b) const

{

return Complex(x - b.x, y - b.y);

}

Complex operator + (const Complex &b) const

{

return Complex(x + b.x, y + b.y);

}

Complex operator \* (const Complex &b) const

{

return Complex(x \* b.x - y \* b.y, x \* b.y + y \* b.x);

}

};

// 进行FFT和IFFT前的反转变换

// 位置i和（i二进制反转后的位置）互换

// len必须去2的幂

void change(Complex y[], int len)

{

int i, j, k;

for (i = 1, j = len / 2; i < len - 1; i++)

{

if (i < j)

{

swap(y[i], y[j]);

}

// 交换护卫小标反转的元素，i < j保证交换一次

// i做正常的+1，j左反转类型的+1，始终保持i和j是反转的

k = len / 2;

while (j >= k)

{

j -= k;

k /= 2;

}

if (j < k)

{

j += k;

}

}

return ;

}

// FFT

// len必须为2 ^ k形式

// on == 1时是DFT，on == -1时是IDFT

void fft(Complex y[], int len, int on)

{

change(y, len);

for (int h = 2; h <= len; h <<= 1)

{

Complex wn(cos(-on \* 2 \* PI / h), sin(-on \* 2 \* PI / h));

for (int j = 0; j < len; j += h)

{

Complex w(1, 0);

for (int k = j; k < j + h / 2; k++)

{

Complex u = y[k];

Complex t = w \* y[k + h / 2];

y[k] = u + t;

y[k + h / 2] = u - t;

w = w \* wn;

}

}

}

if (on == -1)

{

for (int i = 0; i < len; i++)

{

y[i].x /= len;

}

}

}

### FWT

/\*

\* FWT(快速沃尔什变化)-Xor

\* MOD:1e9 + 7, INV\_2:2关于MOD的逆元

\* N:2的整次幂(不够就向上取整)

\*/

typedef long long ll;

const int MOD = 1e9 + 7;

const int INV\_2 = 5e8 + 4;

inline void FWT(int c[], int N, int tf\_utf) // tf\_utf 1:tf; 0:utf

{

for (int i = 1; i < N; i <<= 1)

{

int tmp = i << 1;

for (int j = 0; j < N; j += tmp)

{

for (int k = 0; k < i; k++)

{

int x = c[j + k], y = c[j + k + i];

if (tf\_utf)

{

c[j + k] = x + y;

if (c[j + k] >= MOD)

{

c[j + k] -= MOD;

}

c[j + k + i] = x - y;

if (c[j + k + i] < 0)

{

c[j + k + i] += MOD;

}

}

else

{

c[j + k] = (ll)(x + y) \* INV\_2 % MOD;

c[j + k + i] = (ll)(x - y + MOD) \* INV\_2 % MOD;

}

}

}

}

}

### 整数划分

##### 整数划分（五边形定理）

P(n) = ∑{P(n - k(3k - 1) / 2 + P(n - k(3k + 1) / 2 | k ≥ 1}   
n < 0时，P(n) = 0, n = 0时, P(n) = 1即可

// 划分元素可重复任意次

#define f(x) (((x) \* (3 \* (x) - 1)) >> 1)

#define g(x) (((x) \* (3 \* (x) + 1)) >> 1)

const int MAXN = 1e5 + 10;

const int MOD = 1e9 + 7;

int n, ans[MAXN];

int main()

{

scanf("%d", &n);

ans[0] = 1;

for (int i = 1; i <= n; ++i)

{

for (int j = 1; f(j) <= i; ++j)

{

if (j & 1)

{

ans[i] = (ans[i] + ans[i - f(j)]) % MOD;

}

else

{

ans[i] = (ans[i] - ans[i - f(j)] + MOD) % MOD;

}

}

for (int j = 1; g(j) <= i; ++j)

{

if (j & 1)

{

ans[i] = (ans[i] + ans[i - g(j)]) % MOD;

}

else

{

ans[i] = (ans[i] - ans[i - g(j)] + MOD) % MOD;

}

}

}

printf("%d\n", ans[n]);

return 0;

}

##### 整数划分（五边形定理拓展）

F(n, k) = P(n) - 划分元素重复次数≥k次的情况。

// 问一个数n能被拆分成多少种情况

// 且要求拆分元素重复次数不能≥k

const int MOD = 1e9 + 7;

const int MAXN = 1e5 + 10;

int ans[MAXN];

// 此函数求ans[]效率比上一个代码段中求ans[]效率高很多

void init()

{

memset(ans, 0, sizeof(ans));

ans[0] = 1;

for (int i = 1; i < MAXN; ++i)

{

ans[i] = 0;

for (int j = 1; ; j++)

{

int tmp = (3 \* j - 1) \* j / 2;

if (tmp > i)

{

break;

}

int tmp\_ = ans[i - tmp];

if (tmp + j <= i)

{

tmp\_ = (tmp\_ + ans[i - tmp - j]) % MOD;

}

if (j & 1)

{

ans[i] = (ans[i] + tmp\_) % MOD;

}

else

{

ans[i] = (ans[i] - tmp\_ + MOD) % MOD;

}

}

}

return ;

}

int solve(int n, int k)

{

int res = ans[n];

for (int i = 1; ; i++)

{

int tmp = k \* i \* (3 \* i - 1) / 2;

if (tmp > n)

{

break;

}

int tmp\_ = ans[n - tmp];

if (tmp + i \* k <= n)

{

tmp\_ = (tmp\_ + ans[n - tmp - i \* k]) % MOD;

}

if (i & 1)

{

res = (res - tmp\_ + MOD) % MOD;

}

else

{

res = (res + tmp\_) % MOD;

}

}

return res;

}

int main(int argc, const char \* argv[])

{

init();

int T, n, k;

cin >> T;

while (T--)

{

cin >> n >> k;

cout << solve(n, k) << '\n';

}

return 0;

}

### A^B约数之和

##### A^B约数之和对mod取模

/\*

\* 求A^B的约数之和对MOD取模

\* 需要素数筛选和合数分解的算法，需要先调用getPrime();

\* 参考《合数相关》

\* 1+p+p^2+p^3+...+p^n

\*/

const int MOD = 1000000;

long long pow\_m(long long a, long long n)

{

long long ret = 1;

long long tmp = a % MOD;

while(n)

{

if (n & 1)

{

ret = (ret \* tmp) % MOD;

}

tmp = tmp \* tmp % MOD;

n >>= 1;

}

return ret;

}

// 计算1+p＋p^2+...+p^n

long long sum(long long p, long long n)

{

if (p == 0)

{

return 0;

}

if (n == 0)

{

return 1;

}

if (n & 1)

{

return ((1 + pow\_m(p, n / 2 + 1)) % MOD \* sum(p, n / 2) % MOD) % MOD;

}

else

{

return ((1 + pow\_m(p, n / 2 + 1)) % MOD \* sum(p, n / 2 - 1) + pow\_m(p, n / 2) % MOD) % MOD;

}

}

// 返回A^B的约数之和%MOD

long long solve(long long A, long long B)

{

getFactors(A);

long long ans = 1;

for (int i = 0; i < fatCnt; i++)

{

ans \*= sum(factor[i][0], B \* factor[i][1]) % MOD;

ans %= MOD;

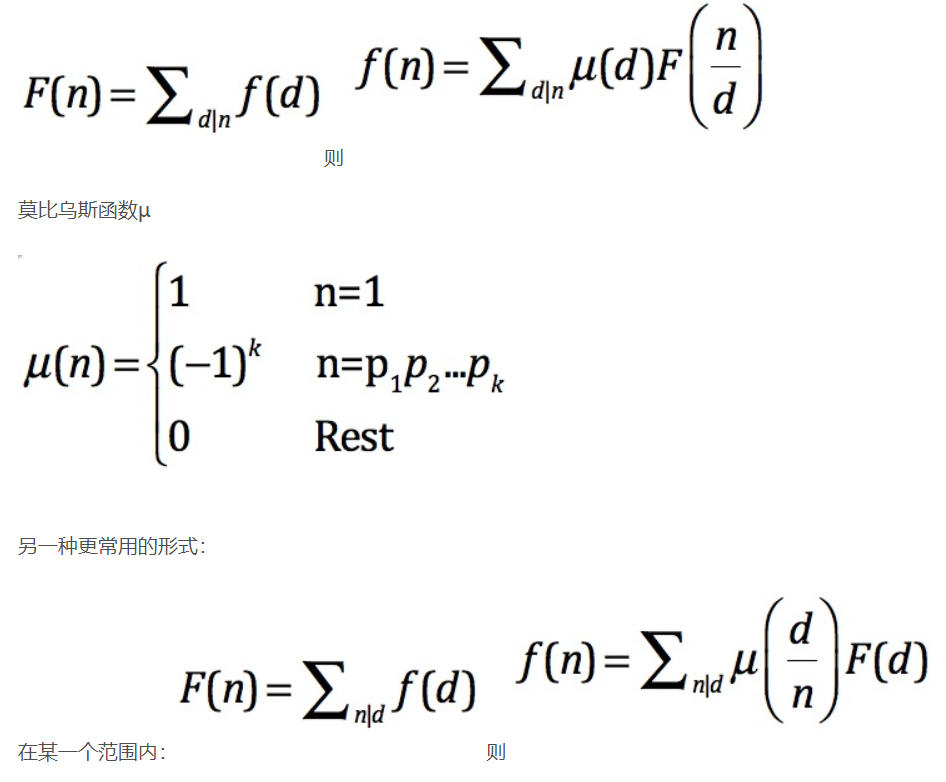
}

return ans;

}

### 莫比乌斯反演

##### 莫比乌斯反演公式



##### 线性筛法求解

/\*

\* 莫比乌斯反演公式

\* ￼￼￼￼￼￼￼线性筛法求解积性函数（莫比乌斯函数）

\*/

const int MAXN = 1000000;

bool check[MAXN + 10];

int prime[MAXN + 10];

int mu[MAXN + 10];

void Moblus()

{

memset(check, false, sizeof(check));

mu[1] = 1;

int tot = 0;

for (int i = 2; i <= MAXN; i++)

{

if (!check[i])

{

prime[tot++] = i;

mu[i] = -1;

}

for (int j = 0; j < tot; j++)

{

if (i \* prime[j] > MAXN)

{

break;

}

check[i \* prime[j]] = true;

if (i % prime[j] == 0)

{

mu[i \* prime[j]] = 0;

break;

}

else

{

mu[i \* prime[j]] = -mu[i];

}

}

}

}

##### 单独求解

int MOD(int a, int b)

{

return a - a / b \* b;

}

int miu(int n)

{

int cnt, k = 0;

for (int i = 2; i \* i <= n; i++)

{

if (MOD(n, i))

{

continue;

}

cnt = 0;

k++;

while (MOD(n, i) == 0)

{

n /= i;

cnt++;

}

if (cnt >= 2)

{

return 0;

}

}

if (n != 1)

{

k++;

}

return MOD(k, 2) ? -1 : 1;

}

### Baby-Step Giant-Step

/\*

\* baby\_step giant \_step

\* a^x = b(mod n) n不要求是素数

\* 求解上式0 ≤ x < n的解

\*/

#define MOD 76543

int hs[MOD];

int head[MOD];

int \_next[MOD];

int id[MOD];

int top;

void insert(int x, int y)

{

int k = x % MOD;

hs[top] = x;

id[top] = y;

\_next[top] = head[k];

head[k] = top++;

return ;

}

int find(int x)

{

int k = x % MOD;

for (int i = head[k]; i != -1; i = \_next[i])

{

if (hs[i] == x)

{

return id[i];

}

}

return -1;

}

long long BSGS(int a, int b, int n)

{

memset(head, -1, sizeof(head));

top = 1;

if (b == 1)

{

return 0;

}

int m = (int)sqrt(n \* 1.0), j;

long long x = 1, p = 1;

for (int i = 0; i < m; i++, p = p \* a % n)

{

insert(p \* b % n, i);

}

for (long long i = m; ; i++)

{

if ((j = find(x = x \* p % n)) != -1)

{

return i - j;

}

if (i > n)

{

break;

}

}

return -1;

}

### 自适应simpson积分

const double eps = 1e-6; // 积分精度

// 被积函数

double F(double x)

{

double ans;

// 被积函数

// ...

// ans = x \* exp(x); // 椭圆为例

return ans;

}

// 三点simpson法，这里要求F是一个全局函数

double simpson(double a, double b)

{

double c = a + (b - a) / 2;

return (F(a) + 4 \* F(c) + F(b)) \* (b - a) / 6;

}

// 自适应simpson公式（递归过程），已知整个区间[a, b]上的三点simpson指A

double asr(double a, double b, double eps, double A)

{

double c = a + (b - a) / 2;

double L = simpson(a, c), R = simpson(c, b);

if (fabs(L + R - A) <= 15 \* eps)

{

return L + R + (L + R - A) / 15.0;

}

return asr(a, c, eps / 2, L) + asr(c, b, eps / 2, R);

}

// 自适应simpson公式（主过程）

double asr(double a, double b, double eps)

{

return asr(a, b, eps, simpson(a, b));

}

int main(int argc, const char \* argv[])

{

// std::cout << asr(1, 2, eps) << '\n';

return 0;

}

### 多项式求根

##### 多项式求根（牛顿法）

/\*

\* 牛顿法解多项式的根

\* 输入:多项式系数c[],多项式度数n,求在[a,b]间的根

\* 输出:根 要求保证[a,b]间有根

\*/

double fabs(double x)

{

return (x < 0) ? -x : x;

}

double f(int m, double c[], double x)

{

int i;

double p = c[m];

for (i = m; i > 0; i--)

{

p = p \* x + c[i - 1];

}

return p;

}

int newton(double x0, double \*r, double c[], double cp[], int n, double a, double b, double eps)

{

int MAX\_ITERATION = 1000;

int i = 1;

double x1, x2, fp, eps2 = eps / 10.0;

x1 = x0;

while (i < MAX\_ITERATION)

{

x2 = f(n, c, x1);

fp = f(n - 1, cp, x1);

if ((fabs(fp) < 0.000000001) && (fabs(x2) > 1.0))

{

return 0;

}

x2 = x1 - x2 / fp;

if (fabs(x1 - x2) < eps2)

{

if (x2 < a || x2 > b)

{

return 0;

}

\*r = x2;

return 1;

}

x1 = x2;

i++;

}

return 0;

}

double Polynomial\_Root(double c[], int n, double a, double b, double eps)

{

double \*cp;

int i;

double root;

cp = (double \*)calloc(n, sizeof(double));

for (i = n - 1; i >= 0; i--)

{

cp[i] = (i + 1) \* c[i + 1];

}

if (a > b)

{

root = a;

a = b;

b = root;

}

if ((!newton(a, &root, c, cp, n, a, b, eps)) && (!newton(b, &root, c, cp, n, a, b, eps)))

{

newton((a + b) \* 0.5, &root, c, cp, n, a, b, eps);

}

free(cp);

if (fabs(root) < eps)

{

return fabs(root);

}

else

return root;

}

### 星期问题

##### 基姆拉尔森公式：

##### W = (D + 2 \* M + 3 \* (M + 1) \ 5 + Y + Y \ 4 - Y \ 100 + Y \ 400) Mod 7

基姆拉尔森公式的计算结果是0，1，2，3，4，5，6 七种可能；   
结果的对应关系：   
0：星期一   
1：星期二   
2：星期三   
3：星期四   
4：星期五   
5：星期六   
6：星期日

/\*

\* 已知1752年9月3日是Sunday，并且日期控制在1700年2月28日后

\*/

char name[][15] = { "monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday"};

int main()

{

int d, m, y, a;

printf("Day: ");

scanf("%d", &d);

printf("Month: ");

scanf("%d", &m);

printf("Year: ");

scanf("%d", &y);

// 1月2月当作前一年的13,14月

if (m == 1 || m == 2)

{

m += 12;

y--;

}

// 判断是否在1752年9月3日之前,实际上合并在一起倒更加省事

if ((y < 1752) || (y == 1752 && m < 9) || (y == 1752 && m == 9 && d < 3))

{

// 因为日期控制在1700年2月28日后，所以不用考虑整百年是否是闰年

a = (d + 2 \* m + 3 \* (m + 1) / 5 + y + y / 4 + 5) % 7;

}

else

{

// 这里需要考虑整百年是否是闰年的情况

a = (d + 2 \* m + 3 \* (m + 1) / 5 + y + y / 4 - y / 100 + y / 400) % 7; // 实际上这个可以当做公式背下来

}

printf("it's a %s\n", name[a]);

return 0;

}

### 斐波那契额数列

##### 矩阵原理单独求解

/\*

\* 求斐波那契数列第N项，模MOD

\*/

#define mod(a, m) ((a) % (m) + (m)) % (m)

const int MOD = 1e9 + 9;

struct MATRIX

{

long long a[2][2];

};

MATRIX a;

long long f[2];

void ANS\_Cf(MATRIX a)

{

f[0] = mod(a.a[0][0] + a.a[1][0], MOD);

f[1] = mod(a.a[0][1] + a.a[1][1], MOD);

return ;

}

MATRIX MATRIX\_Cf(MATRIX a, MATRIX b)

{

MATRIX ans;

int k;

for (int i = 0; i < 2; i++)

{

for (int j = 0; j < 2; j++)

{

ans.a[i][j] = 0;

k = 0;

while (k < 2)

{

ans.a[i][j] += a.a[k][i] \* b.a[j][k];

ans.a[i][j] = mod(ans.a[i][j], MOD);

++k;

}

}

}

return ans;

}

MATRIX MATRIX\_Pow(MATRIX a, long long n)

{

MATRIX ans;

ans.a[0][0] = 1;

ans.a[1][1] = 1;

ans.a[0][1] = 0;

ans.a[1][0] = 0;

while (n)

{

if (n & 1)

{

ans = MATRIX\_Cf(ans, a);

}

n = n >> 1;

a = MATRIX\_Cf(a, a);

}

return ans;

}

int main()

{

long long n;

while (cin >> n)

{

if (n == 1)

{

cout << '1' << '\n';

continue;

}

a.a[0][0] = a.a[0][1] = a.a[1][0] = 1;

a.a[1][1] = 0;

a = MATRIX\_Pow(a, n - 2);

ANS\_Cf(a);

cout << f[0] << '\n';

}

return 0;

}

### 1/n循环节长度

/\*

\* 求1/i的循环节长度的最大值，i<=n

\*/

const int MAXN = 1005;

int res[MAXN]; // 循环节长度

int main()

{

memset(res, 0, sizeof(res));

int i, temp, j, n;

for (temp = 1; temp <= 1000; temp++)

{

i = temp;

while (i % 2 == 0)

{

i /= 2;

}

while (i % 5 == 0)

{

i /= 5;

}

n = 1;

for (j = 1; j <= i; j++)

{

n \*= 10;

n %= i;

if (n == 1)

{

res[temp] = j;

break;

}

}

}

int max\_re;

while (cin >> n)

{

max\_re = 1;

for (i = 1; i <= n; i++)

{

if (res[i] > res[max\_re])

{

max\_re = i;

}

}

cout << max\_re << endl;

}

return 0;

}

### 矩阵相关

##### 矩阵乘法

/\*

\* 矩阵乘法 n\*n矩阵乘法

\*/

#define MAXN 111

#define mod(x) ((x) % MOD)

#define MOD 1000000007

#define LL long long

int n;

struct mat

{

int m[MAXN][MAXN];

};

// 矩阵乘法

mat operator \* (mat a, mat &b)

{

mat ret;

memset(ret.m, 0, sizeof(ret.m));

for (int k = 0; k < n; k++)

{

for (int i = 0; i < n; i++)

{

if (a.m[i][k])

{

for (int j = 0; j < n; j++)

{

ret.m[i][j] = mod(ret.m[i][j] + (LL)a.m[i][k] \* b.m[k][j]);

}

}

}

}

return ret;

}

##### 矩阵乘法+判等

/\*

\* AB == C ???

\*/

struct Matrix

{

Type mat[MAXN][MAXN];

int n, m;

Matrix()

{

n = m = MAXN;

memset(mat, 0, sizeof(mat));

}

Matrix(const Matrix &a)

{

set\_size(a.n, a.m);

memcpy(mat, a.mat, sizeof(a.mat));

}

Matrix & operator = (const Matrix &a)

{

set\_size(a.n, a.m);

memcpy(mat, a.mat, sizeof(a.mat));

return \*this;

}

void set\_size(int row, int column)

{

n = row;

m = column;

}

friend Matrix operator \* (const Matrix &a, const Matrix &b)

{

Matrix ret;

ret.set\_size(a.n, b.m);

for (int i = 0; i < a.n; ++i)

{

for (int k = 0; k < a.m; ++k)

{

if (a.mat[i][k])

{

for (int j = 0; j < b.m; ++j)

{

if (b.mat[k][j])

{

ret.mat[i][j] = ret.mat[i][j] + a.mat[i][k] \* b.mat[k][j];

}

}

}

}

}

return ret;

}

friend bool operator == (const Matrix &a, const Matrix &b)

{

if (a.n != b.n || a.m != b.m)

{

return false;

}

for (int i = 0; i < a.n; ++i)

{

for (int j = 0; j < a.m; ++j)

{

if (a.mat[i][j] != b.mat[i][j])

{

return false;

}

}

}

return true;

}

};

##### 矩阵快速幂

/\*

\* 矩阵快速幂 n\*n矩阵的x次幂

\*/

#define MAXN 111

#define mod(x) ((x) % MOD)

#define MOD 1000000007

#define LL long long

int n;

struct mat

{

int m[MAXN][MAXN];

} unit; // 单元矩阵

// 矩阵乘法

mat operator \* (mat a, mat &b)

{

mat ret;

memset(ret.m, 0, sizeof(ret.m));

for (int k = 0; k < n; k++)

{

for (int i = 0; i < n; i++)

{

if (a.m[i][k])

{

for (int j = 0; j < n; j++)

{

ret.m[i][j] = mod(ret.m[i][j] + (LL)a.m[i][k] \* b.m[k][j]);

}

}

}

}

return ret;

}

void init\_unit()

{

for (int i = 0; i < MAXN; i++)

{

unit.m[i][i] = 1;

}

return ;

}

mat pow\_mat(mat a, LL n)

{

mat ret = unit;

while (n)

{

if (n & 1)

{

// n--;

ret = ret \* a;

}

n >>= 1;

a = a \* a;

}

return ret;

}

int main()

{

LL x;

init\_unit();

while (cin >> n >> x)

{

mat a;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

cin >> a.m[i][j];

}

}

a = pow\_mat(a, x); // a矩阵的x次幂

// 输出矩阵

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (j + 1 == n)

{

cout << a.m[i][j] << endl;

}

else

{

cout << a.m[i][j] << " ";

}

}

}

}

return 0;

}

### 反素数

##### 求最小的因子个数为n个正整数

typedef unsigned long long ULL;

const ULL INF = ~0ULL;

const int MAXP = 16;

int prime[MAXP] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};

int n;

ULL ans;

void dfs(int dept, ULL tmp, int num, int pre) // 深度/当前值/约数个数/上一个数

{

if (num > n)

{

return;

}

if (num == n && ans > tmp)

{

ans = tmp;

}

for (int i = 1; i <= pre; i++)

{

if (ans / prime[dept] < tmp)

{

break;

}

dfs(dept + 1, tmp \*= prime[dept], num \* (i + 1), i);

}

}

int main()

{

while (cin >> n)

{

ans = INF;

dfs(0, 1, 1, 15);

cout << ans << endl;

}

return 0;

}

##### 求n以内的因子最多的数（不止一个则取最小）

typedef long long ll;

const int MAXP = 16;

const int prime[MAXP] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};

ll n, res, ans;

void dfs(ll cur, ll num, int key, ll pre) // 当前值/当前约数数量/当前深度/上一个数

{

if (key >= MAXP)

{

return ;

}

else

{

if (num > ans)

{

res = cur;

ans = num;

}

else if (num == ans) // 如果约数数量相同，则取较小的数

{

res = min(cur, res);

}

ll i;

for ( i = 1; i <= pre; i++)

{

if (cur <= n / prime[key]) // cur\*prime[key]<=n

{

cur \*= prime[key];

dfs(cur, num \* (i + 1), key + 1, i);

}

else

{

break;

}

}

}

}

void solve()

{

res = 1;

ans = 1;

dfs(1, 1, 0, 15);

cout << res << ' ' << ans << endl;

}

int main(int argc, const char \* argv[])

{

int T;

cin >> T;

while (T--)

{

cin >> n;

solve();

}

return 0;

}

### 容斥

const int MAXN = 1111;

int n;

double ans;

double p[MAXN];

void dfs(int x, int tot, double sum) // dfs(1, 0, ?)

{

if (x == n + 1)

{

if (sum == 0.0)

{

return ;

}

if (tot & 1)

{

ans += 1 / sum; // 公式随意变

}

else

{

ans -= 1 / sum;

}

return ;

}

dfs(x + 1, tot, sum);

dfs(x + 1, tot + 1, sum + p[x]);

}

### 母函数

/\*

\* 母函数

\* c1是保存各项质量砝码可以组合的数目

\* c2是中间量，保存每一次的情况

\*/

const int MAXN = 1e4 + 10;

int n;

int c1[MAXN];

int c2[MAXN];

int main()

{

while (cin >> n)

{

for (int i = 0; i <= n; ++i)

{

c1[i] = 1;

c2[i] = 0;

}

for (int i = 2; i <= n; ++i)

{

for (int j = 0; j <= n; ++j)

{

for (int k = 0; k + j <= n; k += i)

{

c2[j + k] += c1[j];

}

}

for (int j = 0; j <= n; ++j)

{

c1[j] = c2[j];

c2[j] = 0;

}

}

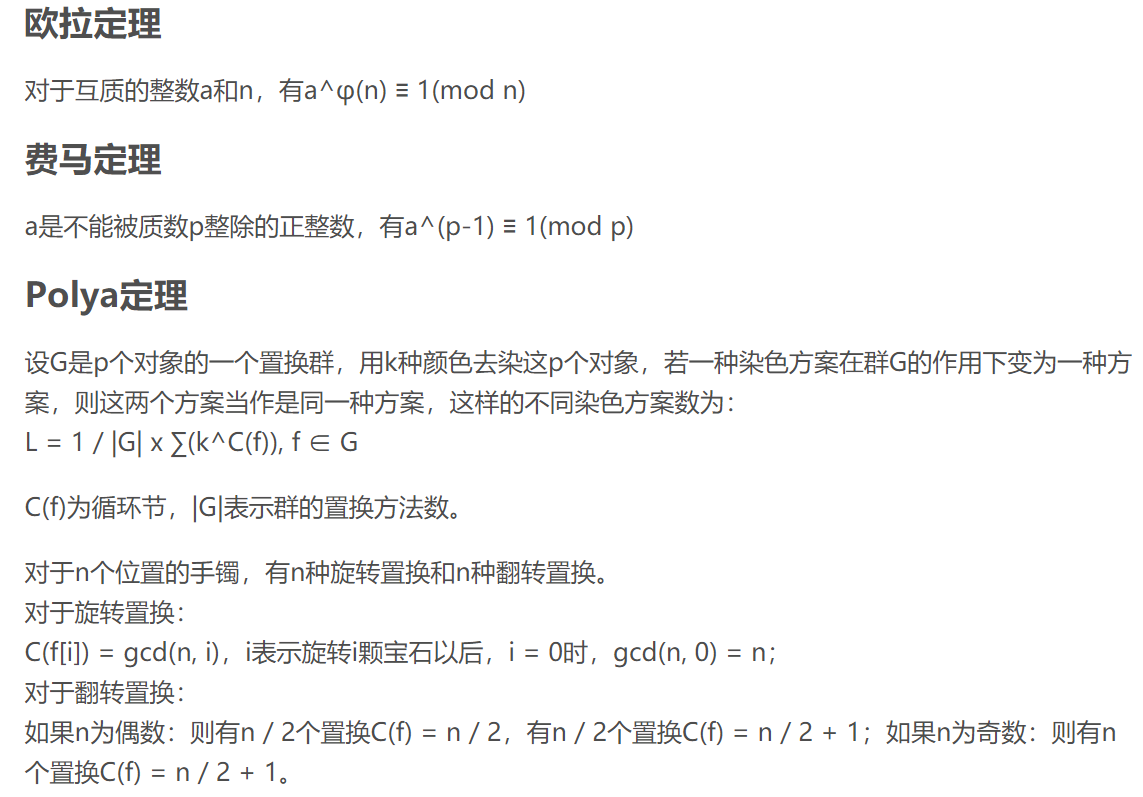
cout << c1[n] << endl;

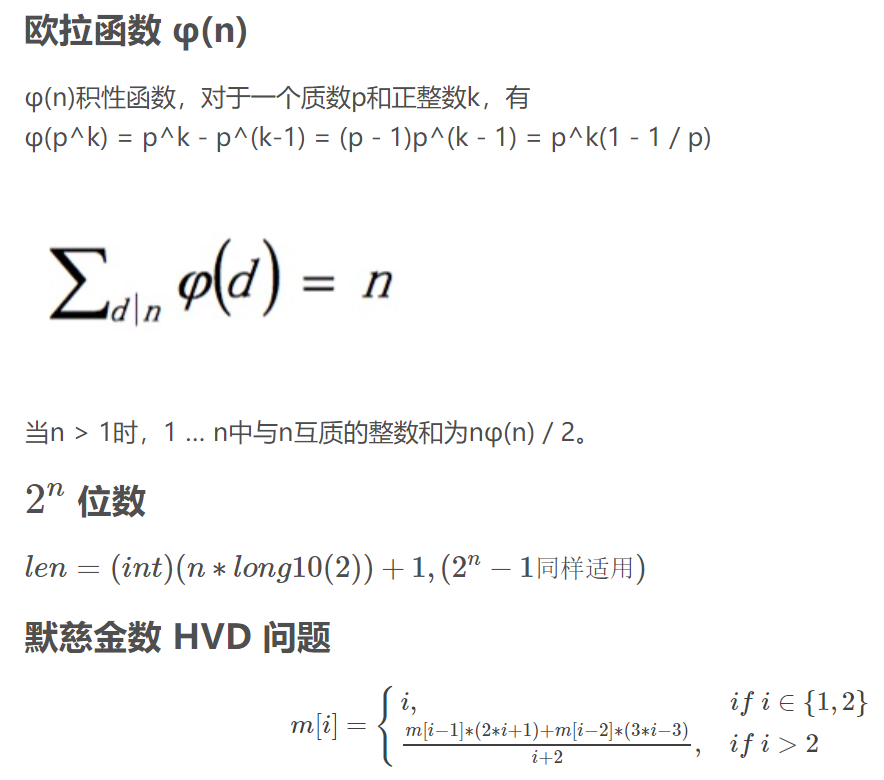
}

return 0;

}

### 数论相关公式





### 素数两种筛法

bool isp[maxn];

int p[maxn], len;

bool isp[100];

void init() {

int m = (int)sqrt(maxn+0.5);

for(int i = 2;i <= m;i++) {

if(!isp[i]) {

for(int j = i\*i;j <= maxn;j += i) {

isp[j] = true;

}

}

}

}

void init() { //推荐这个，较快

isp[0] = isp[1] = true;

for (int i = 2; i < maxn; i++) {

if(!isp[i]) p[++len] = i;

for (int j = 1; j <= len && p[j]\*i < maxn; j++) {

isp[i\*p[j]] = true;

if (i%p[j] == 0) break;

}

}

}

### 埃拉托斯特尼筛法

int prime[maxn];

bool is\_prime[maxn];

int sieve(int n){

int p = 0;

for(int i = 0; i <= n; ++i)

is\_prime[i] = true;

is\_prime[0] = is\_prime[1] = false;

for (int i = 2; i <= n; ++i){ // 注意数组大小是n

if(is\_prime[i]){

prime[p++] = i;

for(int j = i + i; j <= n; j += i) // 轻剪枝，j必定是i的倍数

is\_prime[j] = false;

}

}

return p; // 返回素数个数

}

### 具有最大素因子的整数

样例:36 38 40 42 out:38

int Prime[maxn];

void IsPrime()

{

Prime[1]=1;

for(int i=2;i<=maxn;i++)

{

if(!Prime[i])

{

Prime[i]=i;

for(int j=2\*i;j<=maxn;j+=i)

Prime[j]=i;

}

}

}

### 欧拉函数

在数论，对正整数n，欧拉函数是小于n的正整数中与n互质的数的数目（φ(1)=1）。

int euler\_phi(int n){ //单个值

int m = (int)sqrt(n + 0.5);

int ans = n;

for (int i = 2;i <= m;i++){

if (n%i == 0){ //如果存在素因子

ans = ans/i\*(i-1);

while (n%i == 0) n/=i;

}

}

if(n > 1) ans = ans/n\*(n-1); //考虑n本身

return ans;

}

void phi\_table(int n,int \*phi){ //欧拉表

for (int i = 1;i <= n;i++) phi[i] = i;

for(int i = 2;i <= n;i++){

if(phi[i] == i){ //类似于Eratosthenes筛法这里

for(int j = i;j <= n;j+=i){

phi[j] = phi[j]/i\*(i-1);

}

}

}

}

### 阶乘逆元

fac[0] = 1;

for (int i = 1; i <= maxn; i++)

fac[i] = mod(fac[i - 1] \* i);

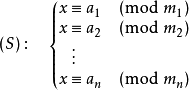
rfac[maxn] = qpow(fac[maxn],MOD - 2);

for (int i = maxn;i > 0; i--)

rfac[i - 1] = mod(rfac[i] \* i);

### 中国剩余定理（CRT拓展）

中国剩余定理给出了以下的一元线性同余方程组：

   
假设整数m1,m2, ... ,mn两两互质，则对任意的整数：a1,a2, ... ,an，方程组 有解,即x，扩展剩余定理就是m1,m2···mn,这几个数不两两互质的情况

ll CRT(ll M){

ll sum=0,tmp,v;

for (int i=1;i<=cnt;i++){

tmp=M/m[i];

v=getInv(tmp,m[i]);

sum=(sum+tmp\*a[i]\*v)%M;

}

return sum;

}

/\*以下是ECRT\*/

bool merge(ll &a1,ll &m1,ll a2,ll m2){

ll c,d,x,a3,m3;

c=a2-a1;d=\_\_gcd(m1,m2);

if (c%d!=0) return false;

c=c/d;m1=m1/d;m2=m2/d;

x=getinv(m1,m2);

x=(x\*c)%m2;

x=x\*(m1\*d)+a1;

m3=m1\*m2\*d;

a3=(x%m3+m3)%m3;

a1=a3;m1=m3;

return true;

}

ll ECRT(){

ll A=a[1],M=r[1];

for (int i=2;i<=n;i++) //无解返回 -1

if (!merge(A,M,a[i],r[i]))

return -1;

return (A%M+M)%M;

}

### Lacus定理模板

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

const int N =1e5;

ll n, m, p, fac[N];

void init()

{

int i;

fac[0] =1;

for(i =1; i <= p; i++)

fac[i] = fac[i-1]\*i % p;

}

ll q\_pow(ll a, ll b)

{

ll ans =1;

while(b)

{

if(b &1) ans = ans \* a % p;

b>>=1;

a = a\*a % p;

}

return ans;

}

ll C(ll n, ll m)

{

if(m > n) return 0;

return fac[n]\*q\_pow(fac[m]\*fac[n-m], p-2) % p;

}

ll Lucas(ll n, ll m )

{

if(m ==0) return 1;

else return (C(n%p, m%p)\*Lucas(n/p, m/p))%p;

}

int main()

{

int t;

scanf("%d", &t);

while(t--)

{

scanf("%I64d%I64d%I64d", &n, &m, &p);

init();

printf("%I64d\n", Lucas(n, m));

}

return 0;

}

#include<iostream>

#include<cstdio>

#define ll long long

using namespace std;

ll a[200005],p,t,n,m;

ll exgcd(ll a,ll b,ll &x,ll &y){//扩展欧几里得

if(b){

ll d=exgcd(b,a%b,y,x);

y-=a/b\*x;

return d;

}

x=1;y=0;return a;

}

ll inv(ll x,ll mod){//求x在模mod下的逆元

ll anx,any;exgcd(x,mod,anx,any);return (anx+mod)%mod;

}

ll C(ll n,ll m){//求C(n,m)%p n<=10^5 m<=10^5

if(m>n)return 0;

return a[n]\*inv(a[m],p)%p\*inv(a[n-m],p)%p;

}

ll lucas(ll n,ll m){//lucas定理求 C(n,m)%p n,m可以很大

if(!m)return 1;

return C(n%p,m%p)\*lucas(n/p,m/p)%p;

}

void prep(){//预处理n!%p

a[0]=1;

for(ll i=1;i<=p;i++){

a[i]=i\*a[i-1]%p;

}

}

void work(){

scanf("%lld%lld%lld",&n,&m,&p);

prep();

printf("%lld\n",lucas(n+m,n));

}

int main(){

scanf("%lld",&t);

while(t--){

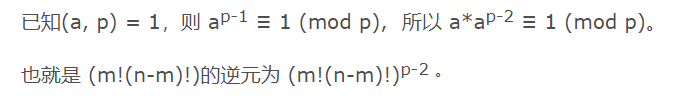
work();

}

return 0;

}

### 费马小定理



# String 字符串

### 编辑距离

编辑距离，又称Levenshtein距离（也叫做Edit Distance），是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

#include <iostream>

#include <cstring>

using namespace std;

typedef long long LL;

const int N = 1e3 + 5;

int T, cas = 0;

int n, m;

int dp[N][N];

char s[N], t[N];

int main()

{

while (scanf("%s%s", s, t) != EOF)

{

int n = (int)strlen(s), m = (int)strlen(t);

for (int i = 0; i <= n; i++)

{

dp[i][0] = i;

}

for (int i = 0; i <= m; i++)

{

dp[0][i] = i;

}

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= m; j++)

{

dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1;

dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + (s[i - 1] != t[j - 1]));

}

}

printf("%d\n", dp[n][m]);

}

return 0;

}

### KMP算法

##### KMP\_Pre

/\*

\* next[]的含义，x[i - next[i]...i - 1] = x[0...next[i] - 1]

\* next[i]为满足x[i - z...i - 1] = x[0...z - 1]的最大z值（就是x的自身匹配）

\*/

void KMP\_Pre(char x[], int m, int next[])

{

int i, j;

j = next[0] = -1;

i = 0;

while (i < m)

{

while (-1 != j && x[i] != x[j])

{

j = next[j];

}

next[++i] = ++j;

}

return ;

}

##### PreKMP

/\*

\* kmpNext[]的意思：next'[i] = next[next[...[next[i]]]]

\* （直到next'[i] < 0或者x[next'[i]] != x[i]）

\* 这样的预处理可以快一些

\*/

void preKMP(char x[], int m, int kmpNext[])

{

int i, j;

j = kmpNext[0] = -1;

i = 0;

while (i < m)

{

while (-1 != j && x[i] != x[j])

{

j = kmpNext[j];

}

if (x[++i] == x[++j])

{

kmpNext[i] = kmpNext[j];

}

else

{

kmpNext[i] = j;

}

}

return ;

}

##### KMP\_Count

/\*

\* 此函数与上述两个函数中的任意一个搭配使用（即调用上述两个函数中的任意一个）

\* 返回x在y中出现的次数，可以重叠

\*/

int next[10010];

int KMP\_Count(char x[], int m, char y[], int n)

{

// x是模式串，y是主串

int i, j;

int ans = 0;

// preKMP(x, m, next);

KMP\_Pre(x, m, next);

i = j = 0;

while (i < n)

{

while (-1 != j && y[i] != x[j])

{

j = next[j];

}

i++, j++;

if (j >= m)

{

ans++;

j = next[j];

}

}

return ans;

}

### 拓展KMP

/\*

\* 扩展KMP

\* next[i]:x[i...m-1]的最长公共前缀

\* extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀

\*/

void preEKMP(char x[], int m, int next[])

{

next[0] = m;

int j = 0;

while (j + 1 < m && x[j] == x[j + 1])

{

j++;

}

next[1] = j;

int k = 1;

for (int i = 2; i < m; i++)

{

int p = next[k] + k - 1;

int L = next[i - k];

if (i + L < p + 1)

{

next[i] = L;

}

else

{

j = std::max(0, p - i + 1);

while (i + j < m && x[i + j] == x[j])

{

j++;

}

next[i] = j;

k = i;

}

}

return ;

}

void EKMP(char x[], int m, char y[], int n, int next[], int extend[])

{

preEKMP(x, m, next);

int j = 0;

while (j < n && j < m && x[j] == y[j])

{

j++;

}

extend[0] = j;

int k = 0;

for (int i = 1; i < n; i++)

{

int p = extend[k] + k - 1;

int L = next[i - k];

if (i + L < p + 1)

{

extend[i] = L;

}

else

{

j = std::max(0, p - i + 1);

while (i + j < n && j < m && y[i + j] == x[j])

{

j++;

}

extend[i] = j;

k = i;

}

}

return ;

}

### 最短公共祖先

将KMP进行略微的改动，依然是查找匹配段，要求要么一个串包含另一个串，要么一个串的前缀等于另一个串的后缀。

/\*

\* The shortest common superstring of 2 strings S1 and S2 is

\* a string S with|the minimum number of characters which

\* contains both S1 and S2 as a sequence of consecutive characters.

\*/

const int N = 1000010;

char a[2][N];

int fail[N];

inline int max(int a, int b)

{

return (a > b) ? a : b;

}

int kmp(int &i, int &j, char\* str, char\* pat)

{

int k;

memset(fail, -1, sizeof(fail));

for (i = 1; pat[i]; ++i)

{

for (k = fail[i - 1]; k >= 0 && pat[i] != pat[k + 1]; k = fail[k]);

if (pat[k + 1] == pat[i])

{

fail[i] = k + 1;

}

}

i = j = 0;

while (str[i] && pat[j])

{

if (pat[j] == str[i])

{

i++;

j++;

}

else if (j == 0)

{

i++; // 第一个字符匹配失败，从str下一个字符开始

}

else

{

j = fail[j - 1] + 1;

}

}

if (pat[j])

{

return -1;

}

else

{

return i - j;

}

}

int main(int argc, const char \* argv[])

{

int T;

scanf("%d", &T);

while (T--)

{

int i, j, l1 = 0, l2 = 0;

cin >> a[0] >> a[1];

int len1 = (int)strlen(a[0]), len2 = (int)strlen(a[1]), val;

val = kmp(i, j, a[1], a[0]); // a[1]在前

if (val != -1)

{

l1 = len1;

}

else

{

// printf("i:%d, j:%d\n", i, j);

if (i == len2 && j - 1 >= 0 && a[1][len2 - 1] == a[0][j - 1])

{

l1 = j;

}

}

val = kmp(i, j, a[0], a[1]); // a[0]在前

if (val != -1)

{

l2 = len2;

}

else

{

// printf("i:%d, j:%d\n", i, j);

if (i == len1 && j - 1 >= 0 && a[0][len1 - 1] == a[1][j - 1])

{

l2 = j;

}

}

// printf("l1:%d,l2:%d\n",l1,l2);

printf("%d\n", len1 + len2 - max(l1, l2));

}

return 0;

}

### Karp-Rabin算法

##### 字符串匹配

/\*

\* hash(w[0 ... m - 1]) =

\* (w[0] \* 2 ^ (m - 1) + ... + w[m - 1] \* 2 ^ 0) % q;

\* hash(w[j + 1 ... j + m]) =

\* rehash(y[j], y[j + m], hash(w[j ... j + m - 1]);

\* rehash(a, b, h) = ((h - a \* 2 ^ (m - 1)) \* 2 + b) % q;

\* 可以用q = 2 ^ 32简化%运算

\*/

#define REHASH(a, b, h) (((h - (a) \* b) << 1) + b)

int krmatch(char \*x, int m, char \*y, int n)

{

//search x in y

int d, hx, hy, i, j;

for (d = i = 1; i < m; i++)

{

d = (d << 1);

}

for (hy = hx = i = 0; i < m; i++)

{

hx = ((hx << 1) + x[i]);

hy = ((hy << 1) + y[i]);

}

for (j = 0; j <= n - m; j++)

{

if (hx == hy && memcmp(x, y + j, m) == 0)

{

return j;

}

hy = REHASH(y[j], y[j + m], hy);

}

return 0; //理论上不会背执行，全部都应该从上一个return返回

}

##### 字符块匹配

/\*

\* Text: n \* m matrix;

\* Pattern: x \* y matrix;

\*/

//#define uint unsigned int // C++中自带

const int A = 1024, B = 128;

const uint E = 27;

char text[A][A];

char patt[B][B];

uint ht, hp;

uint pw[B \* B];

uint hor[A];

uint ver[A][A];

int n, m, x, y;

void init()

{

int i, j = B \* B;

for (i = 1, pw[0] = 1; i < j; i++)

{

pw[i] = pw[i - 1] \* E;

}

return ;

}

void hash()

{

int i, j;

for (i = 0; i < n; i++)

{

for (j = 0, hor[i] = 0; j < y; j++)

{

hor[i] \*= pw[x];

hor[i] += text[i][j] - 'a';

}

}

for (j = 0; j < m; j++)

{

for (i = 0, ver[0][j] = 0; i < x; i++)

{

ver[0][j] \*= E;

ver[0][j] += text[i][j] - 'a';

}

for (i = 1; i <= n - x; i++)

{

ver[i][j] = (ver[i - 1][j] - (text[i - 1][j] - 'a') \* pw[x - 1]) \* E + text[i + x - 1][j] - 'a';

}

}

for (j = 0, ht = hp = 0; j < y; j++)

{

for (i = 0; i < x; i++)

{

ht \*= E;

ht += text[i][j] - 'a';

hp \*= E;

hp += patt[i][j] - 'a';

}

}

return ;

}

void read()

{

int i;

std::cin >> n >> m;

for (i = 0; i < n; i++)

{

std::cin >> text[i];

}

for (i = 0; i < x; i++)

{

std::cin >> patt[i];

}

return ;

}

int solve()

{

if (n == 0 || m == 0 || x == 0 || y == 0)

{

return 0;

}

int i, j, cnt = 0;

uint t;

for (i = 0; i <= n - x; i++)

{

for (j = 0, t = ht; j <= m - y; j++)

{

if (t == hp)

{

cnt++;

}

t = (t - ver[i][j] \* pw[y \* x - x]) \* pw[x] + ver[i][j + y];

}

ht = (ht - hor[i] \* pw[x - 1]) \* E + hor[i + x];

}

return cnt;

}

int main(int argc, const char \* argv[])

{

int T;

init();

for (std::cin >> T; T; T--)

{

read();

hash();

std::cout << solve() << '\n';

}

return 0;

}

### Manacher最长回文子串

/\*

\* 求最长回文子串

\*/

const int MAXN = 110010;

char A[MAXN \* 2];

int B[MAXN \* 2];

void Manacher(char s[], int len)

{

int l = 0;

A[l++] = '$'; //0下标存储为其他字符

A[l++] = '#';

for (int i = 0; i < len; i++)

{

A[l++] = s[i];

A[l++] = '#';

}

A[l] = 0; //空字符

int mx = 0;

int id = 0;

for (int i = 0; i < l; i++)

{

B[i] = mx > i ? std::min(B[2 \* id - i], mx - i) : 1;

while (A[i + B[i]] == A[i - B[i]])

{

B[i]++;

}

if (i + B[i] > mx)

{

mx = i + B[i];

id = i;

}

}

return ;

}

/\*

\* abaaba

\* i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

\* A: $ # a # b # a # a # b # a # '\0'

\* B: 1 1 2 1 4 1 2 7 2 1 4 1 2 1 //以第i个为中心的回文半径（包括第i个）

\*/

char s[MAXN];

int main(int argc, const char \* argv[])

{

while (std::cin >> s)

{

int len = (int)strlen(s);

Manacher(s, len);

int ans = 0;

for (int i = 0; i < 2 \* len + 2; i++) //两倍长度并且首位插有字符，所以i < 2 \* len + 2

{

ans = std::max(ans, B[i] - 1);

}

std::cout << ans << std::endl;

}

return 0;

}

### strstr函数

/\*

\* strstr函数

\* 功能：在串中查找指定字符串的第一次出现

\* 用法：char \*strstr(char \*strOne, char \*strTwo);

\* 据说strstr函数和KMP的算法效率差不多

\*/

int main(int argc, const char \* argv[])

{

char strOne[] = "Borland International";

char strTwo[] = "nation";

char \*ptr;

ptr = strstr(strOne, strTwo);

std::cout << ptr << '\n';

return 0;

}

PS: 输出结果为”national”。

### AC自动机

/\*

\* 求目标串中出现了几个模式串

\*/

struct Trie

{

int next[500010][26], fail[500010], end[500010];

int root, L;

int newnode()

{

for (int i = 0; i < 26; i++)

{

next[L][i] = -1;

}

end[L++] = 0;

return L - 1;

}

void init()

{

L = 0;

root = newnode();

}

void insert(char buf[])

{

int len = (int)strlen(buf);

int now = root;

for (int i = 0; i < len; i++)

{

if (next[now][buf[i] - 'a'] == -1)

{

next[now][buf[i] - 'a'] = newnode();

}

now = next[now][buf[i] - 'a'];

}

end[now]++;

}

void build()

{

queue<int>Q;

fail[root] = root;

for (int i = 0; i < 26; i++)

{

if (next[root][i] == -1)

{

next[root][i] = root;

}

else

{

fail[next[root][i]] = root;

Q.push(next[root][i]);

}

}

while (!Q.empty())

{

int now = Q.front();

Q.pop();

for (int i = 0;i < 26;i++)

{

if (next[now][i] == -1)

{

next[now][i] = next[fail[now]][i];

}

else

{

fail[next[now][i]]=next[fail[now]][i];

Q.push(next[now][i]);

}

}

}

}

int query(char buf[])

{

int len = (int)strlen(buf);

int now = root;

int res = 0;

for (int i = 0; i < len; i++)

{

now = next[now][buf[i] - 'a'];

int temp = now;

while (temp != root)

{

res += end[temp];

end[temp] = 0;

temp = fail[temp];

}

}

return res;

}

void debug()

{

for (int i = 0; i < L; i++)

{

printf("id = %3d,fail = %3d,end = %3d,chi = [", i, fail[i], end[i]);

for (int j = 0; j < 26; j++)

{

printf("%2d", next[i][j]);

}

printf("]\n");

}

}

};

char buf[1000010];

Trie ac;

int main()

{

int T;

int n;

scanf("%d", &T);

while(T--)

{

scanf("%d", &n);

ac.init();

for (int i = 0; i < n; i++)

{

scanf("%s", buf);

ac.insert(buf);

}

ac.build();

scanf("%s", buf);

printf("%d\n", ac.query(buf));

}

return 0;

}

### 后缀数组

##### DA算法

/\*

\* suffix array

\* 倍增算法 O(n\*logn)

\* 待排序数组长度为n,放在0~n-1中,在最后面补一个0

\* da(str, sa, rank, height, n, m);

\* 例如:

\* n = 8;

\* num[] = { 1, 1, 2, 1, 1, 1, 1, 2, $ }; 注意num最后一位为0,其他大于0

\* rank[] = { 4, 6, 8, 1, 2, 3, 5, 7, 0 }; rank[0~n-1]为有效值,rank[n]必定为0无效值

\* sa[] = { 8, 3, 4, 5, 0, 6, 1, 7, 2 }; sa[1~n]为有效值,sa[0]必定为n是无效值

\* height[]= { 0, 0, 3, 2, 3, 1, 2, 0, 1 }; height[2~n]为有效值

\* 稍微改动可以求最长公共前缀，需要注意两串起始位置相同的情况

\* 另外需要注意的是部分数组需要开两倍空间大小

\*/

const int MAXN = 20010;

int t1[MAXN];

int t2[MAXN];

int c[MAXN]; // 求SA数组需要的中间变量,不需要赋值

// 待排序的字符串放在s数组中,从s[0]到s[n-1],长度为n,且最大值小于m,

// 除s[n-1]外的所有s[i]都大于0,r[n-1]=0

// 函数结束以后结果放在sa数组中

bool cmp(int \*r, int a, int b, int l)

{

return r[a] == r[b] && r[a + l] == r[b + l];

}

void da(int str[], int sa[], int rank[], int height[], int n, int m)

{

n++;

int i, j, p, \*x = t1, \*y = t2; // 第一轮基数排序,如果s的最大值很大,可改为快速排序

for (i = 0; i < m; i++)

{

c[i] = 0;

}

for (i = 0; i < n; i++)

{

c[x[i] = str[i]]++;

}

for (i = 1; i < m; i++)

{

c[i] += c[i-1];

}

for (i = n - 1; i >= 0; i--)

{

sa[--c[x[i]]] = i;

}

for (j = 1; j <= n; j <<= 1)

{

p = 0;

// 直接利用sa数组排序第二关键字

for (i = n - j; i < n; i++)

{

y[p++] = i; // 后面的j个数第二关键字为空的最小

}

for (i = 0; i < n; i++)

{

if (sa[i] >= j)

{

y[p++] = sa[i] - j; // 这样数组y保存的就是按照第二关键字排序的结果

}

}

// 基数排序第一关键字

for (i = 0; i < m; i++)

{

c[i] = 0;

}

for (i = 0; i < n; i++)

{

c[x[y[i]]]++;

}

for (i = 1; i < m; i++)

{

c[i] += c[i - 1];

}

for (i = n - 1; i >= 0; i--)

{

sa[--c[x[y[i]]]] = y[i]; // 根据sa和x数组计算新的x数组

}

swap(x, y);

p = 1;

x[sa[0]] = 0;

for (i = 1; i < n; i++)

{

x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;

}

if (p >= n)

{

break;

}

m = p; // 下次基数排序的最大值

}

int k = 0;

n--;

for (i = 0; i <= n; i++)

{

rank[sa[i]] = i;

}

for (i = 0; i < n; i++)

{

if (k)

{

k--;

}

j = sa[rank[i] - 1];

while (str[i + k] == str[j + k])

{

k++;

}

height[rank[i]] = k;

}

}

int \_rank[MAXN], height[MAXN];

int RMQ[MAXN];

int mm[MAXN];

int best[20][MAXN];

void initRMQ(int n)

{

mm[0] = -1;

for (int i = 1; i <= n; i++)

{

mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

}

for (int i = 1; i <= n; i++)

{

best[0][i] = i;

}

for (int i = 1; i <= mm[n]; i++)

{

for (int j = 1; j + (1 << i) - 1 <= n; j++)

{

int a = best[i - 1][j];

int b = best[i - 1][j + (1 << (i - 1))];

if (RMQ[a] < RMQ[b])

{

best[i][j] = a;

}

else

{

best[i][j] = b;

}

}

}

}

int askRMQ(int a, int b)

{

int t;

t = mm[b - a + 1];

b -= (1 << t) - 1;

a = best[t][a];

b = best[t][b];

return RMQ[a] < RMQ[b] ? a : b;

}

int lcp(int a, int b)

{

a = \_rank[a];

b = \_rank[b];

if (a > b)

{

swap(a,b);

}

return height[askRMQ(a + 1, b)];

}

char str[MAXN];

int r[MAXN];

int sa[MAXN];

int main()

{

while (scanf("%s", str) == 1)

{

int len = (int)strlen(str);

int n = 2 \* len + 1;

for (int i = 0; i < len; i++)

{

r[i] = str[i];

}

for (int i = 0; i < len; i++)

{

r[len + 1 + i] = str[len - 1 - i];

}

r[len] = 1;

r[n] = 0;

da(r, sa, \_rank, height, n, 128);

for (int i = 1; i <= n; i++)

{

RMQ[i]=height[i];

}

initRMQ(n);

int ans = 0, st = 0;

int tmp;

for (int i = 0; i < len; i++)

{

tmp = lcp(i, n - i); // 偶对称

if (2 \* tmp > ans)

{

ans = 2 \* tmp;

st = i - tmp;

}

tmp=lcp(i, n - i - 1); // 奇数对称

if (2 \* tmp - 1 > ans)

{

ans = 2 \* tmp - 1;

st = i - tmp + 1;

}

}

str[st + ans] = 0;

printf("%s\n", str + st);

}

return 0;

}

##### DC3算法

da[]和str[]数组都要开大三倍，相关数组也是三倍

/\*

\* 后缀数组

\* DC3算法,复杂度O(n)

\* 所有的相关数组都要开三倍

\*/

const int MAXN = 2010;

#define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))

#define G(x) ((x) < tb ? (x) \* 3 + 1 : ((x)-tb) \* 3 + 2)

int wa[MAXN \* 3], wb[MAXN \* 3], wv[MAXN \* 3], wss[MAXN \* 3];

int c0(int \*r, int a, int b)

{

return r[a] == r[b] && r[a + 1] == r[b + 1] && r[a + 2] == r[b + 2];

}

int c12(int k, int \*r, int a, int b)

{

if(k == 2)

{

return r[a] < r[b] || (r[a] == r[b] && c12(1, r, a + 1, b + 1));

}

else

{

return r[a] < r[b] || (r[a] == r[b] && wv[a + 1] < wv[b + 1]);

}

}

void sort(int \*r, int \*a, int \*b, int n, int m)

{

int i;

for (i = 0; i < n; i++)

{

wv[i] = r[a[i]];

}

for (i = 0; i < m; i++)

{

wss[i] = 0;

}

for (i = 0; i < n; i++)

{

wss[wv[i]]++;

}

for (i = 1; i < m; i++)

{

wss[i] += wss[i - 1];

}

for (i = n - 1; i >= 0; i--)

{

b[--wss[wv[i]]] = a[i];

}

}

void dc3(int \*r, int \*sa, int n, int m)

{

int i, j, \*rn = r + n;

int \*san = sa + n, ta = 0, tb = (n+1)/3, tbc = 0, p;

r[n] = r[n+1] = 0;

for (i = 0; i < n; i++)

{

if (i % 3 != 0)

{

wa[tbc++] = i;

}

}

sort(r + 2, wa, wb, tbc, m);

sort(r + 1, wb, wa, tbc, m);

sort(r, wa, wb, tbc, m);

for (p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++)

{

rn[F(wb[i])] = c0(r, wb[i - 1], wb[i]) ? p - 1 : p++;

}

if (p < tbc)

{

dc3(rn, san, tbc, p);

}

else

{

for (i = 0; i < tbc; i++)

{

san[rn[i]] = i;

}

}

for (i = 0; i < tbc; i++)

{

if (san[i] < tb)

{

wb[ta++] = san[i] \* 3;

}

}

if (n % 3 == 1)

{

wb[ta++] = n - 1;

}

sort(r, wb, wa, ta, m);

for (i = 0; i < tbc; i++)

{

wv[wb[i] = G(san[i])] = i;

}

for (i = 0, j = 0, p = 0; i < ta && j < tbc; p++)

{

sa[p] = c12(wb[j] % 3, r, wa[i], wb[j]) ? wa[i++] : wb[j++];

}

for (; i < ta; p++)

{

sa[p] = wa[i++];

}

for (; j < tbc; p++)

{

sa[p] = wb[j++];

}

}

// str和sa也要三倍

void da(int str[], int sa[], int rank[], int height[], int n,int m)

{

for (int i = n; i < n \* 3; i++)

{

str[i] = 0;

}

dc3(str, sa, n+1, m);

int i, j, k = 0;

for (i = 0; i <= n; i++)

{

rank[sa[i]] = i;

}

for (i = 0; i < n; i++)

{

if(k)

{

k--;

}

j = sa[rank[i] - 1];

while (str[i + k] == str[j + k])

{

k++;

}

height[rank[i]] = k;

}

}

### 后缀自动机

const int CHAR = 26;

const int MAXN = 250010;

struct SAM\_Node

{

SAM\_Node \*fa, \*next[CHAR];

int len;

int id, pos;

SAM\_Node(){}

SAM\_Node(int \_len)

{

fa = 0;

len = \_len;

memset(next, 0, sizeof(next));

}

};

SAM\_Node SAM\_node[MAXN \* 2], \*SAM\_root, \*SAM\_last;

int SAM\_size;

SAM\_Node \*newSAM\_Node(int len)

{

SAM\_node[SAM\_size] = SAM\_Node(len);

SAM\_node[SAM\_size].id = SAM\_size;

return &SAM\_node[SAM\_size++];

}

SAM\_Node \*newSAM\_Node(SAM\_Node \*p)

{

SAM\_node[SAM\_size] = \*p; SAM\_node[SAM\_size].id = SAM\_size;

return &SAM\_node[SAM\_size++];

}

void SAM\_init()

{

SAM\_size = 0;

SAM\_root = SAM\_last = newSAM\_Node(0);

SAM\_node[0].pos = 0;

}

void SAM\_add(int x, int len)

{

SAM\_Node \*p = SAM\_last, \*np = newSAM\_Node(p->len+1);

np->pos = len;

SAM\_last = np;

for (; p && !p->next[x]; p = p->fa)

{

p->next[x] = np;

}

if (!p)

{

np->fa = SAM\_root;

return;

}

SAM\_Node \*q = p->next[x];

if (q->len == p->len + 1)

{

np->fa = q;

return ;

}

SAM\_Node \*nq = newSAM\_Node(q);

nq->len = p->len + 1;

q->fa = nq;

np->fa = nq;

for(;p && p->next[x] == q; p = p->fa)

p->next[x] = nq;

}

void SAM\_build(char \*s)

{

SAM\_init();

int len = (int)strlen(s);

for (int i = 0; i < len; i++)

{

SAM\_add(s[i] - 'a', i + 1);

}

}

/\*

// 加入串后进行拓扑排序

char str[MAXN];

int topocnt[MAXN];

SAM\_Node \*topsam[MAXN \* 2];

int n = (int)strlen(str);

SAM\_build(str);

memset(topocnt, 0, sizeof(topocnt));

for (int i = 0; i < SAM\_size; i++)

{

topocnt[SAM\_node[i].len]++;

}

for (int i = 1; i <= n; i++)

{

topocnt[i] += topocnt[i-1];

}

for (int i = 0; i < SAM\_size; i++)

{

topsam[--topocnt[SAM\_node[i].len]] = &SAM\_node[i];

}

\*/

// 多串的建立:

// 多串的建立,注意SAM\_init()的调用

//void SAM\_build(char \*s)

//{

// int len = (int)strlen(s);

// SAM\_last = SAM\_root;

// for (int i = 0; i < len; i++)

// {

// if (!SAM\_last->next[s[i] - '0'] || !(SAM\_last->next[s[i] - '0']->len == i+1))

// {

// SAM\_add(s[i] - '0',i+1);

// }

// else

// {

// SAM\_last = SAM\_last->next[s[i] - '0'];

// }

// }

//}

### 字符串HASH

/\*

\* 字符串 Hash

\* 注意：mod选择足够大的质数（至少大于字符串个数）

\*/

unsigned int hashA(char \*url, int mod)

{

unsigned int n = 0;

char \*b = (char \*)&n;

for (int i = 0; url[i]; i++)

{

b[i % 4] ^= url[i];

}

return n % mod;

}

unsigned int hashB(char \*url, int mod)

{

unsigned int h = 0;

unsigned int g;

while (\*url)

{

h = (h << 4) + \*url++;

g = h & 0xF0000000;

if (g)

{

h ^= (g >> 24);

}

h &= ~g;

}

return h % mod;

}

unsigned int hashC(char \*p, int prime = 25013)

{

unsigned int h = 0;

unsigned int g;

for (; \*p; p++)

{

h = (h << 4) + \*p;

g = h & 0xF0000000;

if (g)

{

h ^= (g >> 24);

h ^= g;

}

}

return h % prime;

}

### BM算法改进的算法：Sunday Algorithm

BM算法优于KMP

SUNDAY

算法描述:字符串查找算法中,最著名的两个是KMP算法(Knuth-Morris-Pratt)和BM算法(Boyer-Moore)。两个算法在最坏情 况下均具有线性的查找时间。但是在实用上,KMP算法并不比最简单的c库函数strstr()快多少,而BM算法则往往比KMP算法快上3-5倍。但是BM算法还不是最快的算法,这里介绍一种比BM算法更快一些的查找算法。例如我们要在”substring searching algorithm”查找”search”,刚开始时,把子串与文本左边对齐:

substring searching algorithm search

结果在第二个字符处发现不匹配,于是要把子串往后移动。但是该移动多少呢? 这就是各种算法各显神通的地方了,最简单的做法是移动一个字符位置;KMP是利用已经匹配部分的信息来移动;BM算法是做反向比较,并根据已经匹配的部分来确定移动量。这里要介绍的方法是看紧跟在当前子串之后的那个字符(第一个字符串中的’i’)。显然,不管移动多少,这个字符是肯定要参加下一步的比较的,也就是说,如果下一步匹配到了,这个字符必须在子串内。所以,可以移动子串,使子串中的最右边的这个字符与它对齐。现在子串’search’中并不存在’i’,则说明可以直接跳过一大片,从’i’之后的那个字符开始作下一步的比较,如下:

substring searching algorithm search

比较的结果,第一个字符就不匹配,再看子串后面的那个字符,是’r’,它在子串中出现在倒数第三位,于是把子串向后移动三位,使两个’r’对齐,如下:

substring searching algorithm search

这次匹配成功了!回顾整个过程,我们只移动了两次子串就找到了匹配位置, 是不是很神啊?!可以证明,用这个算法,每一步的移动量都比BM算法要大,所以肯定比BM算法更快。

void SUNDAY(char \*text, char \*patt)

{

size\_t temp[256];

size\_t \*shift = temp;

size\_t i, patt\_size = strlen(patt), text\_size = strlen(text);

cout << "size : " << patt\_size << endl;

for(i = 0; i < 256; i++)

{

\*(shift+i) = patt\_size + 1;

}

for(i = 0; i < patt\_size; i++)

{

\*(shift + (unsigned char)(\*(patt+i))) = patt\_size-i; // shift['s']=6步,shitf['e']=5以此类推

}

size\_t limit = text\_size - patt\_size + 1;

for(i = 0; i < limit; i += shift[text[i + patt\_size]])

{

if(text[i] == \*patt)

{

char \*match\_text = text + i + 1;

size\_t match\_size = 1;

do // 输出所有匹配的位置

{

if(match\_size == patt\_size)

{

cout << "the NO. is " << i << endl;

}

}

while((\*match\_text++) == patt[match\_size++]);

}

}

cout << endl;

}

int main(void)

{

char text[100] = "substring searching algorithm search";

char patt[10] = "search";

SUNDAY(text, patt);

return 0;

}

# Graph图论模板

### 最短路

##### Dijkstra 单源最短路 邻接矩阵形式

/\*

\* 单源最短路径，Dijkstra算法，邻接矩阵形式，复杂度为O(n^2)

\* 求出源beg到所有点的最短路径，传入图的顶点数和邻接矩阵cost[][]

\* 返回各点的最短路径lowcost[]，路径pre[]，pre[i]记录beg到i路径上的父节点，pre[beg] = -1

\* 可更改路径权类型，但是权值必须为非负，下标0~n-1

\*/

const int MAXN = 1010;

const int INF = 0x3f3f3f3f; // 表示无穷

bool vis[MAXN];

int pre[MAXN];

void Dijkstra(int cost[][MAXN], int lowcost[], int n, int beg)

{

for (int i = 0; i < n; i++)

{

lowcost[i] = INF;

vis[i] = false;

pre[i] = -1;

}

lowcost[beg] = 0;

for (int j = 0; j < n; j++)

{

int k = -1;

int min = INF;

for (int i = 0; i < n; i++)

{

if (!vis[i] && lowcost[i] < min)

{

min = lowcost[i];

k = i;

}

}

if (k == -1)

{

break;

}

vis[k] = true;

for (int i = 0; i < n; i++)

{

if (!vis[i] && lowcost[k] + cost[k][i] < lowcost[i])

{

lowcost[i] = lowcost[k] + cost[k][i];

pre[i] = k;

}

}

}

}

##### Dijkstra 单源最短路 邻接矩阵形式 双路径信息

/\*

\* 单源最短路径，dijkstra算法，邻接矩阵形式，复杂度为O(n^2)

\* 两点间距离存入map[][],两点间花费存入cost[][]

\* 求出源st到所有点的最短路径及其对应最小花费

\* 返回各点的最短路径lowdis[]以及对应的最小花费lowval[]

\* 可更改路径权类型，但是权值必须为非负，下标1~n

\*/

const int MAXN = 1010;

const int INF = 0x3f3f3f3f;

int n, m;

int lowdis[MAXN];

int lowval[MAXN];

int visit[MAXN];

int map[MAXN][MAXN];

int cost[MAXN][MAXN];

void dijkstra(int st)

{

int temp = 0;

for (int i = 1; i <= n; i++)

{

lowdis[i] = map[st][i];

lowval[i] = cost[st][i];

}

memset(visit, 0, sizeof(visit));

visit[st] = 1;

for (int i = 1; i < n; i++)

{

int MIN = INF;

for (int j = 1; j <= n; j++)

{

if (!visit[j] && lowdis[j] < MIN)

{

temp = j;

MIN = lowdis[j];

}

}

visit[temp] = 1;

for (int j = 1; j <= n; j++)

{

if (!visit[j] && map[temp][j] < INF)

{

if (lowdis[j] > lowdis[temp] + map[temp][j])

{

lowdis[j] = lowdis[temp] + map[temp][j];

lowval[j] = lowval[temp] + cost[temp][j];

}

else if (lowdis[j] == lowdis[temp] + map[temp][j])

{

if (lowval[j] > lowval[temp] + cost[temp][j])

{

lowval[j] = lowval[temp] + cost[temp][j];

}

}

}

}

}

return ;

}

##### Dijkstra 起点Start 结点有权值

#define M 505

const int inf = 0x3f3f3f3f;

int num[M]; // 结点权值

int map[M][M]; // 图的临近矩阵

int vis[M]; // 结点是否处理过

int ans[M]; // 最短路径结点权值和

int dis[M]; // 各点最短路径花费

int n, m, Start, End; // n结点数，m边数，Start起点，End终点

void Dij(int v)

{

ans[v] = num[v];

memset(vis, 0, sizeof(vis));

for (int i = 0; i < n; ++i)

{

if (map[v][i] < inf)

{

ans[i] = ans[v] + num[i];

}

dis[i] = map[v][i];

}

dis[v] = 0;

vis[v] = 1;

for (int i = 1; i < n; ++i)

{

int u = 0, min = inf;

for (int j = 0; j < n; ++j)

{

if (!vis[j] && dis[j] < min)

{

min = dis[j];

u = j;

}

}

vis[u] = 1;

for (int k = 0; k < n; ++k)

{

if (!vis[k] && dis[k] > map[u][k] + dis[u])

{

dis[k] = map[u][k] + dis[u];

ans[k] = ans[u] + num[k];

}

}

for (int k = 0; k < n; ++k)

{

if (dis[k] == map[u][k] + dis[u])

{

ans[k] = max(ans[k], ans[u] + num[k]);

}

}

}

printf("%d %d\n", dis[End], ans[End]); // 输出终点最短路径花费、最短路径结点权值和

}

int main()

{

scanf("%d%d%d%d", &n, &m, &Start, &End);

for (int i = 0; i < n; ++i)

{

scanf("%d", &num[i]);

}

memset(vis, 0, sizeof(vis));

memset(map, 0x3f, sizeof(map));

for (int i = 0; i < m; ++i)

{

int x, y, z;

scanf("%d%d%d", &x, &y, &z);

if (map[x][y] > z)

{

map[x][y] = z;

map[y][x] = z;

}

}

Dij(Start);

return 0;

}

##### Dijkstra 堆优化

/\*

\* 使用优先队列优化Dijkstra算法

\* 复杂度O(ElongE)

\* 注意对vector<Edge> E[MAXN]进行初始化后加边

\*/

const int INF = 0x3f3f3f3f;

const int MAXN = 1000010;

struct qNode

{

int v;

int c;

qNode(int \_v = 0, int \_c = 0) : v(\_v), c(\_c) {}

bool operator < (const qNode &r) const

{

return c > r.c;

}

};

struct Edge

{

int v;

int cost;

Edge(int \_v = 0, int \_cost = 0) : v(\_v), cost(\_cost) {}

};

vector<Edge> E[MAXN];

bool vis[MAXN];

int dist[MAXN]; // 最短路距离

void Dijkstra(int n, int start) // 点的编号从1开始

{

memset(vis, false, sizeof(vis));

memset(dist, 0x3f, sizeof(dist));

priority\_queue<qNode> que;

while (!que.empty())

{

que.pop();

}

dist[start] = 0;

que.push(qNode(start, 0));

qNode tmp;

while (!que.empty())

{

tmp = que.top();

que.pop();

int u = tmp.v;

if (vis[u])

{

continue;

}

vis[u] = true;

for (int i = 0; i < E[u].size(); i++)

{

int v = E[u][i].v;

int cost = E[u][i].cost;

if (!vis[v] && dist[v] > dist[u] + cost)

{

dist[v] = dist[u] + cost;

que.push(qNode(v, dist[v]));

}

}

}

}

void addEdge(int u, int v, int w)

{

E[u].push\_back(Edge(v, w));

}

##### 单源最短路 BellmanFord算法

/\*

\* 单源最短路BellmanFord算法，复杂度O(VE)

\* 可以处理负边权图

\* 可以判断是否存在负环回路，返回true，当且仅当图中不包含从源点可达的负权回路

\* vector<Edge> E;先E.clear()初始化，然后加入所有边

\*/

const int INF = 0x3f3f3f3f;

const int MAXN = 550;

int dist[MAXN];

struct Edge

{

int u;

int v;

int cost;

Edge(int \_u = 0, int \_v = 0, int \_cost = 0) : u(\_u), v(\_v), cost(\_cost){}

};

vector<Edge> E;

bool BellmanFord(int start, int n) // 编号从1开始

{

memset(dist, 0x3f, sizeof(dist));

dist[start] = 0;

for (int i = 1; i < n; i++) // 最多做n - 1次

{

bool flag = false;

for (int j = 0; j < E.size(); j++)

{

int u = E[j].u;

int v = E[j].v;

int cost = E[j].cost;

if (dist[v] > dist[u] + cost)

{

dist[v] = dist[u] + cost;

flag = true;

}

}

if (!flag) // 无负环回路

{

return true;

}

}

for (int j = 0; j < E.size(); j++)

{

if (dist[E[j].v] > dist[E[j].u] + E[j].cost)

{

return false; // 有负环回路

}

}

return true; // 无负环回路

}

##### 单源最短路 SPFA

/\*

\* 时间复杂度O(kE)

\* 队列实现，有时候改成栈实现会更快，较容易修改

\*/

const int MAXN = 1010;

const int INF = 0x3f3f3f3f;

struct Edge

{

int v;

int cost;

Edge(int \_v = 0, int \_cost = 0) : v(\_v), cost(\_cost) {}

};

vector<Edge> E[MAXN];

void addEdge(int u, int v, int w)

{

E[u].push\_back(Edge(v, w));

}

bool vis[MAXN]; // 在队列标志

int cnt[MAXN]; // 每个点的入列队次数

int dist[MAXN];

bool SPFA(int start, int n)

{

memset(vis, false, sizeof(vis));

memset(dist, 0x3f, sizeof(dist));

vis[start] = true;

dist[start] = 0;

queue<int> que;

while (!que.empty())

{

que.pop();

}

que.push(start);

memset(cnt, 0, sizeof(cnt));

cnt[start] = 1;

while (!que.empty())

{

int u = que.front();

que.pop();

vis[u] = false;

for (int i = 0; i < E[u].size(); i++)

{

int v = E[u][i].v;

if (dist[v] > dist[u] + E[u][i].cost)

{

dist[v] = dist[u] + E[u][i].cost;

if (!vis[v])

{

vis[v] = true;

que.push(v);

if (++cnt[v] > n)

{

return false; // cnt[i]为入队列次数，用来判定是否存在负环回路

}

}

}

}

}

return true;

}

##### Floyd算法 邻接矩阵形式

/\*

\* Floyd算法，求从任意节点i到任意节点j的最短路径

\* cost[][]:初始化为INF（cost[i][i]：初始化为0）

\* lowcost[][]:最短路径，path[][]:最短路径（无限制）

\*/

const int MAXN = 100;

int cost[MAXN][MAXN];

int lowcost[MAXN][MAXN];

int path[MAXN][MAXN];

void Floyd(int n)

{

memcpy(lowcost, cost, sizeof(cost));

memset(path, -1, sizeof(path));

for (int k = 0; k < n; k++)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (lowcost[i][j] > (lowcost[i][k] + lowcost[k][j]))

{

lowcost[i][j] = lowcost[i][k] + lowcost[k][j];

path[i][j] = k;

}

}

}

}

return ;

}

##### Floyd算法 点权 + 路径限制

/\*

\* Floyd算法，求从任意节点i到任意节点j的最短路径

\* cost[][]:初始化为INF（cost[i][i]：初始化为0）

\* val[]:点权，lowcost[][]:除起点、终点外的点权之和+最短路径

\* path[][]:路径限制，要求字典序最小的路径，下标1~N

\*/

const int MAXN = 110;

const int INF = 0x1f1f1f1f;

int val[MAXN]; // 点权

int cost[MAXN][MAXN];

int lowcost[MAXN][MAXN];

int path[MAXN][MAXN]; // i~j路径中的第一个结点

void Floyd(int n)

{

memcpy(lowcost, cost, sizeof(cost));

for (int i = 0; i <= n; i++)

{

for (int j = 0; j <= n; j++)

{

path[i][j] = j;

}

}

for (int k = 1; k <= n; k++)

{

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= n; j++)

{

int temp = lowcost[i][k] + lowcost[k][j] + val[k];

if (lowcost[i][j] > temp)

{

lowcost[i][j] = temp;

path[i][j] = path[i][k];

}

else if (lowcost[i][j] == temp && path[i][j] > path[i][k])

{

path[i][j] = path[i][k];

}

}

}

}

return ;

}

### 第K短路

##### Dijkstra

/\*

\* Dijkstra变形，可以证明每个点经过的次数为小于等于K，

\* 所有Dijkstra的数组dist由一维变为二维，记录经过该点

\* 1次、2次......k次的最小值

\* 输出dist[n - 1][k]即可

\*/

int g[1010][1010];

int n, m, x;

const int INF = 0x3f3f3f3f;

int vis[1010];

int dist[1010][20];

int main(int argc, const char \* argv[])

{

while (cin >> n >> m >> x)

{

//初始化

memset(g, 0x3f, sizeof(g));

memset(dist, 0x3f, sizeof(dist));

memset(vis, 0, sizeof(vis));

for (int i = 0; i < m; i++)

{

int p, q, r;

cin >> p >> q >> r;

if (r < g[p][q])

{

g[p][q] = r;

}

}

dist[1][0] = 0;

dist[0][0] = INF;

while (1)

{

int k = 0;

for (int i = 1; i <= n; i++)

{

if (vis[i] < x && dist[i][vis[i]] < dist[k][0])

{

k = i;

}

}

if (k == 0)

{

break;

}

if (k == n && vis[n] == x - 1)

{

break;

}

for (int i = 1; i <= n; i++)

{

if (vis[i] < x && dist[k][vis[k]] + g[k][i] < dist[i][x])

{

dist[i][x] = dist[k][vis[k]] + g[k][i];

for (int j = x; j > 0; j--)

{

if (dist[i][j] < dist[i][j - 1])

{

swap(dist[i][j], dist[i][j - 1]);

}

}

}

}

vis[k]++;

}

if (dist[n][x - 1] < INF)

{

cout << dist[n][x - 1] << endl;

}

else

{

cout << -1 << endl;

}

}

return 0;

}

##### A\*

/\*

\* A\* 估价函数 fi为到当前点走过的路经长度，hi为该点到终点的长度

\* gi = hi + fi

\*/

int n, m, x, ct;

int g[1010][1010];

int gr[1010][1010];

int dist[1010];

int vis[1010];

const int INF = 0x3f3f3f3f;

struct node

{

int id;

int fi;

int gi;

friend bool operator < (node a, node b)

{

if (a.gi == b.gi)

{

return a.fi > b.fi;

}

return a.gi > b.gi;

}

} s[20000010];

int init()

{

memset(dist, 0x3f, sizeof(dist));

for (int i = 0; i <= n; i++)

{

vis[i] = 1;

}

dist[n - 1] = 0;

for (int i = 0; i < n; i++)

{

int k = n;

for (int j = 0; j < n; j++)

{

if (vis[j] && dist[j] < dist[k])

{

k = j;

}

}

if (k == n)

{

break;

}

vis[k] = 0;

for (int j = 0; j < n; j++)

{

if (vis[j] && dist[k] + gr[k][j] < dist[j])

{

dist[j] = dist[k] + gr[k][j];

}

}

}

return 1;

}

int solve()

{

if (dist[0] == INF)

{

return -1;

}

ct = 0;

s[ct].id = 0;

s[ct].fi = 0;

s[ct++].gi = dist[0];

int cnt = 0;

while (ct)

{

int id = s[0].id;

int fi = s[0].fi;

if (id == n - 1)

{

cnt++;

}

if (cnt == x)

{

return fi;

}

pop\_heap(s, s + ct);

ct--;

for (int j = 0; j < n; j++)

{

if (g[id][j] < INF)

{

s[ct].id = j;

s[ct].fi = fi + g[id][j];

s[ct].gi = s[ct].fi + dist[j];

ct++;

push\_heap(s, s + ct);

}

}

}

return -1;

}

int main()

{

while (cin >> n >> m >> x)

{

memset(g, 0x3f, sizeof(g));

memset(gr, 0x3f, sizeof(gr));

for (int i = 0; i < n; i++)

{

int p, q, r;

cin >> p >> q >> r;

p--;

q--;

g[p][q] = g[p][q] <= r ? g[p][q] : r;

gr[q][p] = gr[q][p] <= r ? gr[q][p] : r;

}

init();

cout << solve() << endl;

}

return 0;

}

### 最小生成树（森林）

##### Prim算法

/\*

\* Prim求MST

\* 耗费矩阵cost[][]，初始化为INF，标号从0开始，0 ~ n－1

\* 返回最小生成树的权值，返回-1表示原图不连通

\*/

const int INF = 0x3f3f3f3f;

const int MAXN = 110;

bool vis[MAXN];

int lowc[MAXN];

int cost[MAXN][MAXN];

// 修正cost（添加边）

void updata(int x, int y, int v)

{

cost[x - 1][y - 1] = v;

cost[y - 1][x - 1] = v;

return ;

}

int Prim(int cost[][MAXN], int n) // 0 ~ n - 1

{

int ans = 0;

memset(vis, false, sizeof(vis));

vis[0] = true;

for (int i = 1; i < n; i++)

{

lowc[i] = cost[0][i];

}

for (int i = 1; i < n; i++)

{

int minc = INF;

int p = -1;

for (int j = 0; j < n; j++)

{

if (!vis[j] && minc > lowc[j])

{

minc = lowc[j];

p = j;

}

}

if (minc == INF)

{

return -1; // 原图不连通

}

ans += minc;

vis[p] = true;

for (int j = 0; j < n; j++)

{

if (!vis[j] && lowc[j] > cost[p][j])

{

lowc[j] = cost[p][j];

}

}

}

return ans;

}

##### Kruskal算法

/\*

\* Kruskal算法求MST

\* 对边操作，并排序

\* 切记：初始化赋值问题（tol）

\*/

const int MAXN = 110; // 最大点数

const int MAXM = 10000; // 最大边数

int F[MAXN]; // 并查集使用

struct Edge

{

int u; // 起点

int v; // 终点

int w; // 权值

} edge[MAXM]; // 存储边的信息

int tol; // 边数，加边前赋值为0

void addEdge(int u, int v, int w)

{

edge[tol].u = u;

edge[tol].v = v;

edge[tol++].w = w;

return ;

}

bool cmp(Edge a, Edge b)

{

// 排序函数，将边按照权值从小到大排序

return a.w < b.w;

}

int find(int x)

{

if (F[x] == x)

{

return x;

}

else

{

return F[x] = find(F[x]);

}

}

int Kruskal(int n) // 传入点数，返回最小生成树的权值，如果不连通则返回-1

{

for (int i = 0; i <= n; i++)

{

F[i] = i;

}

sort(edge, edge + tol, cmp);

int cnt = 0; // 计算加入的边数

int ans = 0;

for (int i = 0; i < tol; i++)

{

int u = edge[i].u;

int v = edge[i].v;

int w = edge[i].w;

int tOne = find(u);

int tTwo = find(v);

if (tOne != tTwo)

{

ans += w;

F[tOne] = tTwo;

cnt++;

}

if (cnt == n - 1)

{

break;

}

}

if (cnt < n - 1)

{

return -1; // 不连通

}

else

{

return ans;

}

}

##### MST

/\*

\* Minimal Steiner Tree

\* G(V, E), A是V的一个子集, 求至少包含A中所有点的最小子树.

\* 时间复杂度:O(N^3+N\*2^A\*(2^A+N))

\* INIT: d[][]距离矩阵; id[]置为集合A中点的标号;

\* CALL: steiner(int n, int a);

\* 给4个点对(a1,b1)...(a4,b4),

\* 求min(sigma(dist[ai][bi])),其中重复的路段只能算一次.

\* 这题要找出一个Steiner森林, 最后要对森林中树的个数进行枚举

\*/

#define typec int // type of cost

const typec inf = 0x3f3f3f3f; // max of cost

const typec V = 10010;

const typec A = 10;

int vis[V], id[A]; // id[]: A中点的标号

typec d[V][V], dp[1 << A][V]; // dp[i][v]: 点v到点集i的最短距离

void steiner(int n, int a)

{

int i, j, k, mx, mk = 0, top = (1 << a);

for (k = 0; k < n; k++)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (d[i][j] > d[i][k] + d[k][j])

{

d[i][j] = d[i][k] + d[k][j];

}

}

}

}

for (i = 0; i < a; i++)

{

// vertex: 0 ~ n-1

for (j = 0; j < n; j++)

{

dp[1 << i][j] = d[j][id[i]];

}

}

for (i = 1; i < top; i++)

{

if (0 == (i & (i - 1)))

{

continue;

}

memset(vis, 0, sizeof(vis));

for (k = 0; k < n; k++) // init

{

for (dp[i][k] = inf, j = 1; j < i; j++)

{

if ((i | j) == i && dp[i][k] > dp[j][k] + dp[i - j][k])

{

dp[i][k] = dp[j][k] + dp[i - j][k];

}

}

}

for (j = 0; mx = inf, j < n; j++)

{

// update

for (k = 0; k < n; k++)

{

if (dp[i][k] <= mx && 0 == vis[k])

{

mx = dp[i][mk = k];

}

}

for (k = 0, vis[mk] = 1; k < n; k++)

{

if (dp[i][mk] > dp[i][k] + d[k][mk])

{

dp[i][mk] = dp[i][k] + d[k][mk];

}

}

}

}

return ;

}

int main(int argc, const char \* argv[])

{

int n, a = 8;

int b, z, i, j, k, x = 0, y;

// TODO: read data;

steiner(n, a);

// enum to find the result

for (i = 0, b = inf; z = 0, i < 256; b > z ? b = z : b, i++)

{

for (j = 0; y = 0, j < 4; z += !!y \* dp[y][x], j++)

{

for (k = 0; k < 8; k += 2)

{

if ((i >> k & 3) == j)

{

y += 3 << k, x = id[k];

}

}

}

}

// TODO: cout << b << endl;

return 0;

}

### 次小生成树

O(V^2)

次小生成树可由最小生成树转换一条边得到

只要充分利用上述结论，既得v^2的算法。具体如下：

step1. 先用Prim求出最小生成树T，在Prim的同时，用一个矩阵MAX[u][v]记录在T中连结任意两点u，v的唯一的路中权值最大的那条边的权值。（注意这里），这是很容易做到的，因为Prim是每次增加一个结点s，而已经标好了的结点集合为w，则w中所有的结点到s的路中最大权值的边就是当前加入的这条边，用时O(V^2)；

step2.枚举所有不在T中的边u\_v，加入边u\_v替换权为MAX[u][v]的边，不断更新最小值，即次小生成树，用时O(E)，故总用时O(V^2)

/\*

\* 求最小生成树时，用数组MAX[i][j]表示i到j的最大边权

\* 求完后，直接枚举所有不在MST中的边，替换掉最大边权的边，更新答案

\* 点的编号从0开始

\*/

const int MAXN = 110;

const int INF = 0x3f3f3f3f;

bool vis[MAXN];

int lowc[MAXN];

int pre[MAXN];

int MAX[MAXN][MAXN];

bool used[MAXN][MAXN];

int Prim(int cost[][MAXN], int n)

{

int ans = 0;

memset(vis, false, sizeof(vis));

memset(MAX, 0, sizeof(MAX));

memset(used, false, sizeof(used));

vis[0] = true;

pre[0] = -1;

lowc[0] = 0;

for (int i = 1; i < n; i++)

{

lowc[i] = cost[0][i];

pre[i] = 0;

}

for (int i = 1; i < n; i++)

{

int minc = INF;

int p = -1;

for (int j = 0; j < n; j++)

{

if (!vis[j] && minc > lowc[j])

{

minc = lowc[j];

p = j;

}

}

if (minc == INF)

{

return -1;

}

ans += minc;

vis[p] = true;

used[p][pre[p]] = used[pre[p]][p] = true;

for (int j = 0; j < n; j++)

{

if (vis[j])

{

MAX[j][p] = MAX[p][j] = max(MAX[j][pre[p]], lowc[p]);

}

if (!vis[j] && lowc[j] > cost[p][j])

{

lowc[j] = cost[p][j];

pre[j] = p;

}

}

}

return ans;

}

### 曼哈顿最小生成树

曼哈顿距离：简单说，他指两点之间的横纵坐标的差的绝对值之和。

题意:查找平面上的点的曼哈顿距离最小生成树的第n－k小边的长度，点数在100000以内。

解析: 对于曼哈顿距离的最小生成树，朴素算法需要建立n^(n - 1)条边进行kruskal算法处理，这样子做一定会TLE的。所以需要做特殊的优化，将边数优化为4 \* n条。

这里的优化涉及到一个与曼哈顿相关的性质：以任一一个点为端点，将平面分为八块，每块占45度角，那么在生成树的最优解中，每个块与这个点至多有一条边，即一个点最多分别向八个方向最近的点连接一条边，一条边两个点共用，所以最后边数为4 \* n。

#include <iostream>

#include <algorithm>

const int MAXN = 100010;

const int INF = 0x3f3f3f3f;

struct Point

{

int x;

int y;

int id;

}poi[MAXN];

bool cmp(Point a, Point b)

{

if (a.x != b.x)

{

return a.x < b.x;

}

else

{

return a.y < b.y;

}

}

//树状数组，找y - x大于当前的，但是y + x最小的

struct BIT

{

int minVal;

int pos;

void init()

{

minVal = INF;

pos = -1;

}

}bit[MAXN];

//所有有效边

struct Edge

{

int u;

int v;

int d;

}edge[MAXN << 2];

bool cmpEdge(Edge a, Edge b)

{

return a.d < b.d;

}

int tot;

int n;

int F[MAXN];

int find(int x)

{

if (F[x] == -1)

{

return x;

}

else

{

return F[x] = find(F[x]);

}

}

void addEdge(int u, int v, int d)

{

edge[tot].u = u;

edge[tot].v = v;

edge[tot++].d = d;

return ;

}

int lowbit(int x)

{

return x & (-x); //???

}

//更新bit

void update(int i, int val, int pos)

{

while (i > 0)

{

if (val < bit[i].minVal)

{

bit[i].minVal = val;

bit[i].pos = pos;

}

i -= lowbit(i);

}

return ;

}

//查询[i, m]的最小值位置

int ask(int i, int m)

{

int minVal = INF;

int pos = -1;

while (i <= m)

{

if (bit[i].minVal < minVal)

{

minVal = bit[i].minVal;

pos = bit[i].pos;

}

i += lowbit(i);

}

return pos;

}

int dist(Point a, Point b)

{

return abs(a.x - b.x) + abs(a.y - b.y);

}

void ManhattanMinimumSpanningTree(int n, Point p[])

{

int a[MAXN], b[MAXN];

tot = 0;

for (int dir = 0; dir < 4; dir++)

{

//变换4种坐标

if (dir == 1 || dir == 3)

{

for (int i = 0; i < n; i++)

{

std::swap(p[i].x, p[i].y);

}

}

else if (dir == 2)

{

for (int i = 0; i < n; i++)

{

p[i].x = -p[i].x;

}

}

std::sort(p, p + n, cmp);

for (int i = 0; i < n; i++)

{

a[i] = b[i] = p[i].y - p[i].x;

}

std::sort(b, b + n);

int m = (int)(std::unique(b, b + n) - b);

for (int i = 1; i <= m; i++)

{

bit[i].init();

}

for (int i = n - 1; i >= 0; i--)

{

int pos = (int)(std::lower\_bound(b, b + m, a[i]) - b + 1);

int ans = ask(pos, m);

if (ans != -1)

{

addEdge(p[i].id, p[ans].id, dist(p[i], p[ans]));

}

update(pos, p[i].x + p[i].y, i);

}

}

return ;

}

int solve(int k)

{

ManhattanMinimumSpanningTree(n, poi);

memset(F, -1, sizeof(F));

std::sort(edge, edge + tot, cmpEdge);

for (int i = 0; i < tot; i++)

{

int u = edge[i].u;

int v = edge[i].v;

int tOne = find(u);

int tTwo = find(v);

if (tOne != tTwo)

{

F[tOne] = tTwo;

k--;

if (k == 0)

{

return edge[i].d;

}

}

}

return -1;

}

int main(int argc, const char \* argv[])

{

//freopen("in.txt", "r", stdin);

//freopen("out.txt", "w", stdout);

int k;

while ((std::cin >> n >> k) && n)

{

for (int i = 0; i < n; i++)

{

std::cin >> poi[i].x >> poi[i].y;

poi[i].id = i;

}

std::cout << solve(n - k) << std::endl;

}

return 0;

}

### 欧拉路径

##### 无向图：

连通（不考虑度为0的点），每个顶点度数都为偶数。

/\*

\* SGU 101

\*/

struct Edge

{

int to;

int next;

int index;

int dir;

bool flag;

} edge[220];

int head[10]; //前驱

int tot;

void init()

{

memset(head, -1, sizeof((head)));

tot = 0;

}

void addEdge(int u, int v, int index)

{

edge[tot].to = v;

edge[tot].next = head[u];

edge[tot].index = index;

edge[tot].dir = 0;

edge[tot].flag = false;

head[u] = tot++;

edge[tot].to = u;

edge[tot].next = head[v];

edge[tot].index = index;

edge[tot].dir = 1;

edge[tot].flag = false;

head[v] = tot++;

return ;

}

int du[10];

std::vector<int>ans;

void dfs(int u)

{

for (int i = head[u]; i != -1; i = edge[i].next)

{

if (!edge[i].flag)

{

edge[i].flag = true;

edge[i ^ 1].flag = true;

dfs(edge[i].to);

ans.push\_back(i); //容器尾部插入i

}

}

return ;

}

int main()

{

//freopen("in.txt", "r", stdin);

//freopen("out.txt", "w", stdout);

int n;

while (std::cin >> n)

{

init();

int u, v;

memset(du, 0, sizeof(du));

for (int i = 1; i <= n; i++)

{

std::cin >> u >> v;

addEdge(u, v, i);

du[u]++;

du[v]++;

}

int s = -1;

int cnt = 0;

for (int i = 0; i <= 6; i++)

{

if (du[i] & 1)

{

cnt++;

s = i;

}

if (du[i] > 0 && s == -1)

{

s = i;

}

}

if (cnt != 0 && cnt != 2)

{

std::cout << "No solution" << '\n';

continue;

}

ans.clear();

dfs(s);

if (ans.size() != n)

{

std::cout << "No solution" << '\n';

continue;

}

for (int i = 0; i < ans.size(); i++)

{

printf("%d ", edge[ans[i]].index);

if (edge[ans[i]].dir == 0)

{

std::cout << "-" << '\n';

}

else

{

std::cout << "+" << '\n';

}

}

}

return 0;

}

##### 有向图：

基图连通（把边当成无向边，同样不考虑度为0的点），每个顶点出度等于入度。

##### 混合图：

既有无向边也有有向边，首先是基图连通（不考虑度为0的点），然后需要借助网络流判定。

首先给原图中的每条无向边随便指定一个方向（称为初始定向），将原图改为有向图G’，然后的任务就是改变G’中某些条边的方向（当然是无向边转化来的，愿混合图中的有向边不能动）使其满足每个点的入度等于出度。

设D[i]为G’中（点i的出度－点i的入度），即可发现，在改变G’中边的方向的过程中，任何点的D值的奇偶性都不会发生改变（设将边<i, j>改为<j, i>，则i入度加1出度减1，j入度减1出度佳1，两者之差加2或者减2，奇偶性不变），而最终要求的是每个点的入度等于出度，即每个点的D值都为0，是偶数，姑可得：若初始定向得到的G’中任一个点D值是奇数，那么原图中一定不存在欧拉环。

若初始D值都是偶数，则将G’改装成网络：设立源点S和汇点T，对于每个D[i] > 0的点i，连边<S, i>，容量为D[i]/2；对于每个D[j] < 0的点j，连边<j, T>，容量为-D[j]/2；G’中的每条边在网络中仍保留，容量为i（表示该边最多只能被改变一次方向）。求这个网络的最大流，若S引出的所有边均满流，则原混合图是欧拉图，将网络中所有流量为1的中间边（就是不与S或T关联的边）在G’中改变方向，形成的新图G”一定是有向欧拉图；若S引出的边中有的没有满流，则原混合图不是欧拉图。

##### 欧拉路径

每条边只经过一次，不要求回到起点

##### 无向图：

连通（不考虑度为0的点），每个顶点度数都为偶数或者仅有两个点的度数为奇数。

/\*

\* O(E)

\* INIT:adj[][]置为图的邻接表；cnt[a]为a点的邻接点数

\* CALL:alpath(0); 注意：不要有自向边

\*/

const int V = 10000;

int adj[V][V];

int idx[V][V];

int cnt[V];

int stk[V];

int top = 0;

int path(int v)

{

for (int w; cnt[v] > 0; v = w)

{

stk[top++] = v;

w = adj[v][--cnt[v]];

adj[w][idx[w][v]] = adj[w][--cnt[w]];

//处理的是无向图——边是双向边，删除v->w后，还要处理删除w->v

}

return v;

}

void elpath(int b, int n)

{

int i, j;

for (i = 0; i < n; i++)

{

for (j = 0; j < cnt[i]; j++)

{

idx[i][adj[i][j]] = j;

}

}

std::cout << b;

for (top = 0; path(b) == b && top != 0; )

{

b = stk[--top];

std::cout << '-' << b;

}

std::cout << std::endl;

}

##### 有向图：

基图连通（把边当成无向边，同样不考虑度为0的点），每个顶点出度等于入度或者有且仅有一个点的出度比入度多1，有且仅有一个点的出度比入度少1，其余的出度等于入度。

/\*

\* POJ 2337

\* 给出n个小写字母组成的单词，要求将n个单词连接起来。使得前一个单词的最后一个字母和

\* 后一个单词的第一个字母相同。输出字典序最小解

\*/

struct Edge

{

int to;

int next;

int index;

bool flag;

}edge[2010];

int head[30];

int tot;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

}

void addEdge(int u, int v, int index)

{

edge[tot].to = v;

edge[tot].next = head[u];

edge[tot].index = index;

edge[tot].flag = false;

head[u] = tot++;

return ;

}

std::string str[1010];

int in[30];

int out[30];

int cnt;

int ans[1010];

void dfs(int u)

{

for (int i = head[u]; i != -1; i = edge[i].next)

{

if (!edge[i].flag)

{

edge[i].flag = true;

dfs(edge[i].to);

ans[cnt++] = edge[i].index;

}

}

return ;

}

int main(int argc, const char \* argv[])

{

// freopen("in.txt", "r", stdin);

// freopen("out.txt", "w", stdout);

int T, n;

std::cin >> T;

while (T--)

{

std::cin >> n;

for (int i = 0; i < n; i++)

{

std::cin >> str[i];

}

std::sort(str, str + n); //要输出字典序最小的解，先按照字典序排序

init();

memset(in, 0, sizeof(in));

memset(out, 0, sizeof(out));

int start = 100;

for (int i = n - 1; i >= 0; i--) //字典序大的先加入

{

int u = str[i][0] - 'a';

int v = str[i][str[i].length() - 1] - 'a';

addEdge(u, v, i);

out[u]++;

in[v]++;

if (n < start)

{

start = u;

}

if (v < start)

{

start = v;

}

}

int ccOne = 0;

int ccTwo = 0;

for (int i = 0; i < 26; i++)

{

if (out[i] - in[i] == 1)

{

ccOne++;

start = 1; //如果有一个出度比入度大1的点，就从这个点出发，否则从最小的点出发

}

else if (out[i] - in[i] == -1)

{

ccTwo++;

}

else if (out[i] - in[i] != 0)

{

ccOne = 3;

}

}

if (!((ccOne == 0 && ccTwo == 0) || (ccOne == 1 && ccTwo == 1)))

{

std::cout << "\*\*\*" << '\n';

continue;

}

cnt = 0;

dfs(start);

if (cnt != n) //判断是否连通

{

std::cout << "\*\*\*" << '\n';

continue;

}

for (int i = cnt - 1; i >= 0; i--)

{

std::cout << str[ans[i]];

if (i > 0)

{

std::cout << '.';

}

else

{

std::cout << '\n';

}

}

}

return 0;

}

##### 混合图：

如果存在欧拉回路，一定存在欧拉路径，否则如果有且仅有两个点的（出度－入度）是奇数，那么给这两个点加边，判断是否存在欧拉回路，如果存在就一定存在欧拉路径。

/\*

\* POJ 1637

\* 本题保证了连通，故不需要判断连通，否则要判断连通

\*/

const int MAXN = 210;

const int MAXM = 20100; //最大流ISAP部分

const int INF = 0x3f3f3f3f;

struct Edge

{

int to;

int next;

int cap;

int flow;

}edge[MAXM];

int tol;

int head[MAXN];

int gap[MAXN];

int dep[MAXN];

int pre[MAXN];

int cur[MAXN];

void init()

{

tol = 0;

memset(head, -1, sizeof(head));

return ;

}

void addEdge(int u, int v, int w, int rw = 0)

{

edge[tol].to = v;

edge[tol].cap = w;

edge[tol].next = head[u];

edge[tol].flow = 0;

head[u] = tol++;

edge[tol].to = u;

edge[tol].cap = rw;

edge[tol].next = head[v];

edge[tol].flow = 0;

head[v] = tol++;

return ;

}

int sap(int start, int end, int N)

{

memset(gap, 0, sizeof(gap));

memset(dep, 0, sizeof(dep));

memcpy(cur, head, sizeof(head));

int u = start;

pre[u] = -1;

gap[0] = N;

int ans = 0;

while (dep[start] < N)

{

if (u == end)

{

int MIN = INF;

for (int i = pre[u]; i != -1; i = pre[edge[i ^ 1].to])

{

if (MIN > edge[i].cap - edge[i].flow)

{

MIN = edge[i].cap - edge[i].flow;

}

}

for (int i = pre[u]; i != -1; i = pre[edge[i ^ 1].to])

{

edge[i].flow += MIN;

edge[i ^ 1].flow -= MIN;

}

u = start;

ans += MIN;

continue;

}

bool flag = false;

int v = 0;

for (int i = cur[u]; i != -1; i = edge[i].next)

{

v = edge[i].to;

if (edge[i].cap - edge[i].flow && dep[v] + 1 == dep[u])

{

flag = true;

cur[u] = pre[v] = i;

break;

}

}

if (flag)

{

u = v;

continue;

}

int MIN = N;

for (int i = head[u]; i != -1; i = edge[i].next)

{

if (edge[i].cap - edge[i].flow && dep[edge[i].to] < MIN)

{

MIN = dep[edge[i].to];

cur[u] = i;

}

}

gap[dep[u]]--;

if (!gap[dep[u]])

{

return ans;

}

dep[u] = MIN + 1;

gap[dep[u]]++;

if (u != start)

{

u = edge[pre[u] ^ 1].to;

}

}

return ans;

}

//the end of 最大流部分

int in[MAXN];

int out[MAXN];

int main()

{

//freopen("in.txt", "r", stdin);

//freopen("out.txt", "w", stdout);

int T;

int n, m;

std::cin >> T;

while (T--)

{

std::cin >> n >> m;

init();

int u, v, w;

memset(in, 0, sizeof(in));

memset(out, 0, sizeof(out));

while (m--)

{

std::cin >> u >> v >> w;

out[u]++;

in[v]++;

if (w == 0)

{

addEdge(u, v, 1); //双向

}

}

bool flag = true;

for (int i = 1; i <= n; i++)

{

if (out[i] - in[i] > 0)

{

addEdge(0, i, (out[i] - in[i]) / 2);

}

else if (in[i] - out[i] > 0)

{

addEdge(i, n + 1, (in[i] - out[i]) / 2);

}

if ((out[i] - in[i]) & 1)

{

flag = false;

}

}

if (!flag)

{

std::cout << "impossible" << '\n';

continue;

}

sap(0, n + 1, n + 2);

for (int i = head[0]; i != -1; i = edge[i].next)

{

if (edge[i].cap > 0 && edge[i].cap > edge[i].flow)

{

flag = false;

break;

}

}

if (flag)

{

std::cout << "possible" << '\n';

}

else

{

std::cout << "impossible" << '\n';

}

}

return 0;

}

### DAG的深度优先搜索标记

/\*

\* DAG(有向无环图)的深度优先搜索标记

\* INIT:edge[][]邻接矩阵；pre[], post[], tag全置0

\* CALL:dfsTag(i, n); pre/post:开始/结束时间

\*/

const int V = 1010;

int edge[V][V];

int pre[V];

int post[V];

int tag;

void dfsTag(int cur, int n)

{

//vertex:0 ~ n - 1

pre[cur] = ++tag;

for (int i = 0; i < n; i++)

{

if (edge[cur][i])

{

if (0 == pre[i])

{

std::cout << "Three Edge!" << '\n';

dfsTag(i, n);

}

else

{

if (0 == post[i])

{

std::cout << "Back Edge!" << '\n';

}

else if (pre[i] > pre[cur])

{

std::cout << "Down Edge!" << '\n';

}

else

{

std::cout << "Cross Edge!" << '\n';

}

}

}

}

post[cur] = ++tag;

return ;

}

### 图的割点、桥和双连通分支的基本概念

##### 点连通度与边连通度

在一个无向连通图中,如果有一个顶点集合,删除这个顶点集合,以及这个集合中所有顶点相关联的边以后,原图变成多个连通块,就称这个点集为割点集合。一个图的点连通度的定义为,最小割点集合中的顶点数。 类似的,如果有一个边集合,删除这个边集合以后,原图变成多个连通块,就称这个点集为割边集合。一个图的边连通度的定义为,最小割边集合中的边数。

##### 双连通图、割点与桥

如果一个无向连通图的点连通度大于1,则称该图是点双连通的(point biconnected),简称双连通或重连通。一个图有割点,当且仅当这个图的点连通度为1,则割点集合的唯一元素被称为割点(cut point),又叫关节 点(articulation point)。如果一个无向连通图的边连通度大于1,则称该图是边双连通的(edge biconnected),简称双连通或重连通。一个图有桥,当且仅当这个图的边连通度为 1,则割边集合的唯一元素被称为桥(bridge),又叫关节边 (articulation edge)。

可以看出,点双连通与边双连通都可以简称为双连通,它们之间是有着某种联系的,下文中提到的双连通, 均既可指点双连通,又可指边双连通。

##### 双连通分支

在图G的所有子图G’中，如果G’是双连通的,则称G’为双连通子图。如果一个双连通子图G’它不是任何一个双连通子图的真子集,则G’为极大双连通子图。双连通分支(biconnected component),或重连通分支, 就是图的极大双连通子图。特殊的,点双连通分支又叫做块。

##### 求割点与桥

该算法是R.Tarjan发明的。对图深度优先搜索,定义DFS(u)为u在搜索树(以下简称为树)中被遍历到的次序号。定义Low(u)为u或u的子树中能通过非父子边追溯到的最早的节点,即DFS序号最小的节点。根据定义,则有:Low(u)=Min{DFS(u)DFS(v)(u,v)为后向边(返祖边)等价于DFS(v) < DFS(u)且v不为u的父亲节点Low(v)(u,v)为树枝边(父子边)}一个顶点u是割点,当且仅当满足(1)或(2)(1)u为树根,且u有多于一个子树。(2)u不为树根,且满足存在(u,v)为树枝边(或称父子边,即u为v在搜索树中的父亲),使得 DFS(u) <= Low(v)。一条无向边(u,v)是桥,当且仅当(u,v)为树枝边,且满足DFS(u) < Low(v)。

##### 求双连通分支

下面要分开讨论点双连通分支与边双连通分支的求法。

对于点双连通分支,实际上在求割点的过程中就能顺便把每个点双连通分支求出。建立一个栈,存储当前双连通分支,在搜索图时,每找到一条树枝边或后向边(非横叉边),就把这条边加入栈中。如果遇到某时满 足DFS(u) <= Low(v),说明u是一个割点,同时把边从栈顶一个个取出,直到遇到了边(u,v),取出的这些边与其关联的点,组成一个点双连通分支。割点可以属于多个点双连通分支,其余点和每条边只属于且属于一个点双连通分支。对于边双连通分支,求法更为简单。只需在求出所有的桥以后,把桥边删除,原图变成了多个连通块,则每个连通块就是一个边双连通分支。桥不属于任何一个边双连通分支,其余的边和每个顶点都属于且只属于一个边双连通分支。

##### 构造双连通图

一个有桥的连通图,如何把它通过加边变成边双连通图?

方法为首先求出所有的桥,然后删除这些桥边, 剩下的每个连通块都是一个双连通子图。把每个双连通子图收缩为一个顶点,再把桥边加回来,最后的这个图一定是一棵树,边连通度为1。统计出树中度为1的节点的个数,即为叶节点的个数,记为leaf。则至少在树上添加(leaf + 1) / 2条边,就能使树达到边二连通,所以至少添加的边数就是(leaf + 1) / 2。具体方法为,首先把两个最近公共祖先最远的两个叶节点之间连接一条边,这样可以把这两个点到祖先的路径上所有点收缩到一起,因为一个形成的环一定是双连通的。然后再找两个最近公共祖先最远的两个叶节点,这样一对一对找完,恰好是(leaf + 1) / 2 次,把所有点收缩到了一起。

### 无向图找桥

/\*

\* 无向图找桥

\* INIT: edge[][]邻接矩阵；vis[],pre[],ans[],bridge置0；

\* CALL: dfs(0, -1, 1, n);

\*/

const int V = 1010;

int bridge; //桥

int edge[V][V];

int ans[V];

int pre[V];

int vis[V];

void dfs(int cur, int father, int dep, int n)

{

//vertex: 0 ~ n - 1

if (bridge)

{

return ;

}

vis[cur] = 1;

pre[cur] = ans[cur] = dep;

for (int i = 0; i < n; i++)

{

if (edge[cur][i])

{

if (i != father && 1 == vis[i])

{

if (pre[i] < ans[cur])

{

ans[cur] = pre[i]; //back edge

}

}

if (0 == vis[i]) //tree edge

{

dfs(i, cur, dep + 1, n);

if (bridge)

{

return ;

}

if (ans[i] < ans[cur])

{

ans[cur] = ans[i];

}

if (ans[i] > pre[cur])

{

bridge = 1;

return ;

}

}

}

}

vis[cur] = 2;

return ;

}

### 无向图连通度（割）

/\*

\* INIT: edge[][]邻接矩阵；vis[],pre[],anc[],deg[]置为0；

\* CALL: dfs(0, -1, 1, n);

\* k = deg[0], deg[i] + 1(i = 1...n - 1)为删除该节点后得到的连通图个数

\* 注意: 0作为根比较特殊

\*/

const int V = 1010;

int edge[V][V];

int anc[V];

int pre[V];

int vis[V];

int deg[V];

void dfs(int cur, int father, int dep, int n)

{

//vertex:0 ~ n - 1

int cnt = 0;

vis[cur] = 1;

pre[cur] = anc[cur] = dep;

for (int i = 0; i < n; i++)

{

if (edge[cur][i])

{

if (i != father && 1 == vis[i])

{

if (pre[i] < anc[cur])

{

anc[cur] = pre[i]; //back edge

}

}

if (0 == vis[i]) //tree edge

{

dfs(i, cur, dep + 1, n);

cnt++; //分支个数

if (anc[i] < anc[cur])

{

anc[cur] = anc[i];

}

if ((cur == 0 && cnt > 1) || (cnt != 0 && anc[i] >= pre[cur]))

{

deg[cur]++; //link degree of a vertex

}

}

}

}

vis[cur] = 2;

return ;

}

### 最大团问题

##### DP+DFS

/\*

\* INIT: g[][]邻接矩阵

\* CALL: res = clique(n);

\*/

const int V = 10010;

int g[V][V];

int dp[V];

int stk[V][V];

int mx;

int dfs(int n, int ns, int dep)

{

if (0 == ns)

{

if (dep > mx)

{

mx = dep;

}

return 1;

}

int i, j, k, p, cnt;

for (i = 0; i < ns; i++)

{

k = stk[dep][i];

cnt = 0;

if (dep + n - k <= mx)

{

return 0;

}

if (dep + dp[k] <= mx)

{

return 0;

}

for (j = i + 1; j < ns; j++)

{

p = stk[dep][j];

if (g[k][p])

{

stk[dep + 1][cnt++] = p;

}

}

dfs(n, cnt, dep + 1);

}

return 1;

}

int clique(int n)

{

int i, j, ns;

for (mx = 0, i = n - 1; i >= 0; i--) // vertex: 0 ~ n-1

{

for (ns = 0, j = i + 1; j < n; j++)

{

if (g[i][j])

{

stk[1][ns++] = j;

}

}

dfs(n, ns, 1);

dp[i] = mx;

}

return mx;

}

### 最小树形图

/\*

\* 最小树形图

\* int型

\* 复杂度O(NM)

\* 点从0开始

\*/

const int INF = 0x3f3f3f3f;

const int MAXN = 1010;

const int MAXM = 1000010;

struct Edge

{

int u, v, cost;

};

Edge edge[MAXM];

int pre[MAXN], id[MAXN], visit[MAXN], in[MAXN];

int zhuliu(int root, int n, int m)

{

int res = 0, v;

while (1)

{

memset(in, 0x3f, sizeof(in));

for (int i = 0; i < m; i++)

{

if (edge[i].u != edge[i].v && edge[i].cost < in[edge[i].v])

{

pre[edge[i].v] = edge[i].u;

in[edge[i].v] = edge[i].cost;

}

}

for (int i = 0; i < n; i++)

{

if (i != root && in[i] == INF)

{

return -1; // 不存在最小树形图

}

}

int tn = 0;

memset(id, -1, sizeof(id));

memset(visit, -1, sizeof(visit));

in[root] = 0;

for (int i = 0; i < n; i++)

{

res += in[i];

v = i;

while (visit[v] != i && id[v] == -1 && v != root)

{

visit[v] = i;

v = pre[v];

}

if (v != root && id[v] == -1)

{

for (int u = pre[v]; u != v ; u = pre[u])

{

id[u] = tn;

}

id[v] = tn++;

}

}

if (tn == 0)

{

break; // 没有有向环

}

for (int i = 0; i < n; i++)

{

if (id[i] == -1)

{

id[i] = tn++;

}

}

for (int i = 0; i < m; i++)

{

v = edge[i].v;

edge[i].u = id[edge[i].u];

edge[i].v = id[edge[i].v];

if (edge[i].u != edge[i].v)

{

edge[i].cost -= in[v];

}

}

n = tn;

root = id[root];

}

return res;

}

### 一般图匹配带花树

const int maxn = 300;

int N;

bool G[maxn][maxn];

int match[maxn];

bool InQueue[maxn], InPath[maxn], InBlossom[maxn];

int head, tail;

int Queue[maxn];

int start, finish;

int NewBase;

int father[maxn], Base[maxn];

int Count;

void CreateGraph()

{

int u, v;

memset(G, 0, sizeof(G));

scanf("%d", &N);

while (scanf("%d%d",&u,&v) != EOF)

{

G[u][v] = G[v][u] = 1;

}

}

void Push(int u)

{

Queue[tail++] = u;

InQueue[u] = 1;

}

int Pop()

{

int res = Queue[head++];

return res;

}

int FindCommonAncestor (int u, int v)

{

memset(InPath, 0, sizeof(InPath));

while (true)

{

u = Base[u];

InPath[u] = 1;

if (u == start)

{

break;

}

u = father[match[u]];

}

while (true)

{

v = Base[v];

if (InPath[v])

{

break;

}

v = father[match[v]];

}

return v;

}

void ResetTrace(int u)

{

int v;

while (Base[u] != NewBase)

{

v = match[u];

InBlossom[Base[u]] = InBlossom[Base[v]] = 1;

u = father[v];

if (Base[u] != NewBase)

{

father[u] = v;

}

}

}

void BlossomContract(int u, int v)

{

NewBase = FindCommonAncestor(u, v);

memset(InBlossom, 0, sizeof(InBlossom));

ResetTrace(u);

ResetTrace(v);

if (Base[u] != NewBase)

{

father[u]=v;

}

if (Base[v] != NewBase)

{

father[v]=u;

}

for (int tu=1; tu <= N; tu++)

{

if (InBlossom[Base[tu]])

{

Base[tu] = NewBase;

if (!InQueue[tu])

{

Push(tu);

}

}

}

}

void FindAugmentingPath()

{

memset(InQueue, 0, sizeof(InQueue));

memset(father, 0, sizeof(father));

for (int i = 1; i <= N; i++)

{

Base[i] = i;

}

head = tail = 1;

Push(start);

finish = 0;

while (head < tail)

{

int u = Pop();

for (int v = 1; v <= N; v++)

{

if (G[u][v] && (Base[u] != Base[v]) && match[u] != v)

{

if ((v == start) || ((match[v] > 0) && father[match[v]] > 0))

{

BlossomContract(u, v);

}

else if (father[v] == 0)

{

father[v] = u;

if (match[v] > 0)

{

Push(match[v]);

}

else

{

finish = v;

return ;

}

}

}

}

}

}

void AugmentPath()

{

int u, v, w;

u = finish;

while (u > 0)

{

v = father[u];

w = match[v];

match[v] = u;

match[u] = v;

u = w;

}

}

void Edmonds()

{

memset(match, 0, sizeof(match));

for (int u = 1; u <= N; u++)

{

if (match[u] == 0)

{

start = u;

FindAugmentingPath();

if (finish > 0)

{

AugmentPath();

}

}

}

}

void PrintMatch()

{

Count = 0;

for (int u = 1; u <= N; u++)

{

if (match[u] > 0)

{

Count++;

}

}

printf("%d\n", Count);

for (int u = 1; u <= N; u++)

{

if (u < match[u])

{

printf("%d %d\n", u, match[u]);

}

}

}

int main()

{

CreateGraph();

Edmonds(); // 进行匹配

PrintMatch(); // 输出匹配

return 0;

}

### LCA

##### DFS+ST在线算法

const int MAXN = 10010;

int rmq[2 \* MAXN]; // rmq数组,就是欧拉序列对应的深度序列

struct ST

{

int mm[2 \* MAXN];

int dp[2 \* MAXN][20]; // 最小值对应的下标

void init(int n)

{

mm[0] = -1;

for (int i = 1; i <= n; i++)

{

mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

dp[i][0] = i;

}

for (int j = 1; j <= mm[n]; j++)

{

for (int i = 1; i + (1 << j) - 1 <= n; i++)

{

dp[i][j] = rmq[dp[i][j - 1]] < rmq[dp[i + (1 << (j - 1))][j - 1]] ? dp[i][j - 1] : dp[i + (1 << (j - 1))][j - 1];

}

}

}

int query(int a,int b) // 查询[a,b]之间最小值的下标

{

if (a > b)

{

swap(a, b);

}

int k = mm[b - a + 1];

return rmq[dp[a][k]] <= rmq[dp[b - (1 << k) + 1][k]] ? dp[a][k] : dp[b - (1 << k) + 1][k];

}

};

// 边的结构体定义

struct Edge

{

int to, next;

};

Edge edge[MAXN \* 2];

int tot, head[MAXN];

int F[MAXN \* 2]; // 欧拉序列,就是dfs遍历的顺序,长度为2\*n-1,下标从1开始

int P[MAXN]; // P[i]表示点i在F中第一次出现的位置

int cnt;

ST st;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

}

void addedge(int u, int v) // 加边,无向边需要加两次

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

}

void dfs(int u, int pre, int dep)

{

F[++cnt] = u;

rmq[cnt] = dep;

P[u] = cnt;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v == pre)

{

continue;

}

dfs(v, u, dep + 1);

F[++cnt] = u;

rmq[cnt] = dep;

}

}

void LCA\_init(int root, int node\_num) // 查询LCA前的初始化

{

cnt = 0;

dfs(root, root, 0);

st.init(2 \* node\_num - 1);

}

int query\_lca(int u, int v) // 查询u,v的lca编号

{

return F[st.query(P[u], P[v])];

}

bool flag[MAXN];

int main()

{

int T;

int N;

int u, v;

scanf("%d", &T);

while(T--)

{

scanf("%d", &N);

init();

memset(flag, false, sizeof(flag));

for (int i = 1; i < N; i++)

{

scanf("%d%d", &u, &v);

addedge(u, v);

addedge(v, u);

flag[v] = true;

}

int root;

for (int i = 1; i <= N; i++)

{

if (!flag[i])

{

root = i;

break;

}

}

LCA\_init(root, N);

scanf("%d%d", &u, &v);

printf("%d\n", query\_lca(u, v));

}

return 0;

}

##### Tarjan离线算法

/\*

\* 给出一颗有向树，Q个查询

\* 输出查询结果中每个点出现次数

\* 复杂度O(n + Q);

\*/

const int MAXN = 1010;

const int MAXQ = 500010; // 查询数的最大值

// 并查集部分

int F[MAXN]; // 需要初始化为-1

int find(int x)

{

if (F[x] == -1)

{

return x;

}

return F[x] = find(F[x]);

}

void bing(int u, int v)

{

int t1 = find(u);

int t2 = find(v);

if (t1 != t2)

{

F[t1] = t2;

}

}

bool vis[MAXN]; // 访问标记

int ancestor[MAXN]; // 祖先

struct Edge

{

int to, next;

} edge[MAXN \* 2];

int head[MAXN],tot;

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

}

struct Query

{

int q, next;

int index; // 查询编号

} query[MAXQ \* 2];

int answer[MAXQ]; // 存储最后的查询结果,下标0~Q-1

int h[MAXQ];

int tt;

int Q;

void add\_query(int u, int v, int index)

{

query[tt].q = v;

query[tt].next = h[u];

query[tt].index = index;

h[u] = tt++;

query[tt].q = u;

query[tt].next = h[v];

query[tt].index = index;

h[v] = tt++;

}

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

tt = 0;

memset(h, -1, sizeof(h));

memset(vis, false, sizeof(vis));

memset(F, -1, sizeof(F));

memset(ancestor, 0, sizeof(ancestor));

}

void LCA(int u)

{

ancestor[u] = u;

vis[u] = true;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (vis[v])

{

continue;

}

LCA(v);

bing(u, v);

ancestor[find(u)] = u;

}

for (int i = h[u]; i != -1; i = query[i].next)

{

int v = query[i].q;

if (vis[v])

{

answer[query[i].index] = ancestor[find(v)];

}

}

}

bool flag[MAXN];

int Count\_num[MAXN];

int main()

{

int n;

int u, v, k;

while (scanf("%d", &n) == 1)

{

init();

memset(flag, false, sizeof(flag));

for (int i = 1; i <= n; i++)

{

scanf("%d:(%d)", &u, &k);

while (k--)

{

scanf("%d", &v);

flag[v] = true;

addedge(u,v);

addedge(v,u);

}

}

scanf("%d", &Q);

for (int i = 0; i < Q; i++)

{

char ch;

cin >> ch;

scanf("%d %d)", &u, &v);

add\_query(u, v, i);

}

int root;

for (int i = 1; i <= n; i++)

{

if (!flag[i])

{

root = i;

break;

}

}

LCA(root);

memset(Count\_num, 0, sizeof(Count\_num));

for (int i = 0; i < Q; i++)

{

Count\_num[answer[i]]++;

}

for (int i = 1; i <= n; i++)

{

if (Count\_num[i] > 0)

{

printf("%d:%d\n", i, Count\_num[i]);

}

}

}

return 0;

}

##### 倍增法

/\*

\* LCA在线算法(倍增法)

\*/

const int MAXN = 10010;

const int DEG = 20;

struct Edge

{

int to, next;

} edge[MAXN \* 2];

int head[MAXN], tot;

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

}

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

}

int fa[MAXN][DEG]; // fa[i][j]表示结点i的第2^j个祖先

int deg[MAXN]; // 深度数组

void BFS(int root)

{

queue<int>que;

deg[root] = 0;

fa[root][0] = root;

que.push(root);

while (!que.empty())

{

int tmp = que.front();

que.pop();

for (int i = 1; i < DEG; i++)

{

fa[tmp][i] = fa[fa[tmp][i - 1]][i - 1];

}

for (int i = head[tmp]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v == fa[tmp][0])

{

continue;

}

deg[v] = deg[tmp] + 1;

fa[v][0] = tmp;

que.push(v);

}

}

}

int LCA(int u, int v)

{

if (deg[u] > deg[v])

{

swap(u, v);

}

int hu = deg[u], hv = deg[v];

int tu = u, tv = v;

for (int det = hv-hu, i = 0; det ; det >>= 1, i++)

{

if (det & 1)

{

tv = fa[tv][i];

}

}

if (tu == tv)

{

return tu;

}

for (int i = DEG - 1; i >= 0; i--)

{

if (fa[tu][i] == fa[tv][i])

{

continue;

}

tu = fa[tu][i];

tv = fa[tv][i];

}

return fa[tu][0];

}

bool flag[MAXN];

int main()

{

int T;

int n;

int u, v;

scanf("%d", &T);

while(T--)

{

scanf("%d", &n);

init();

memset(flag, false, sizeof(flag));

for (int i = 1; i < n; i++)

{

scanf("%d%d", &u, &v);

addedge(u, v);

addedge(v, u);

flag[v] = true;

}

int root;

for (int i = 1; i <= n; i++)

{

if (!flag[i])

{

root = i;

break;

}

}

BFS(root);

scanf("%d%d", &u, &v);

printf("%d\n", LCA(u, v));

}

return 0;

}

### 生成树计数

Matrix-Tree 定理(Kirchhoff 矩阵-树定理)

1、G 的度数矩阵 D[G]是一个 n\*n 的矩阵,并且满足:当 i≠j 时,dij=0;当 i=j 时,dij 等于 vi 的度数。

2、G 的邻接矩阵 A[G]也是一个 n\*n 的矩阵, 并且满足:如果 vi、vj 之间有边直接相连,则 aij=1,否则

为 0。

我们定义 G 的 Kirchhoff 矩阵(也称为拉普拉斯算子)C[G]为 C[G]=D[G]-A[G],则 Matrix-Tree 定理可以

描述为:G 的所有不同的生成树的个数等于其 Kirchhoff 矩阵 C[G]任何一个 n-1 阶主子式的行列式的绝对

值。所谓 n-1 阶主子式,就是对于 r(1≤r≤n),将 C[G]的第 r 行、第 r 列同时去掉后得到的新矩阵,用 Cr[G]

表示。

##### 求生成树计数部分代码,计数对10007取模

// 求生成树计数部分代码,计数对10007取模

const int MOD = 10007;

int INV[MOD];

// 求ax = 1(mod m)的x值,就是逆元(0<a<m)

long long inv(long long a, long long m)

{

if (a == 1)

{

return 1;

}

return inv(m % a, m) \* (m - m / a) % m;

}

struct Matrix

{

int mat[330][330];

void init()

{

memset(mat, 0, sizeof(mat));

}

int det(int n) // 求行列式的值模上MOD,需要使用逆元

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

mat[i][j] = (mat[i][j] % MOD + MOD) % MOD;

}

}

int res = 1;

for (int i = 0; i < n; i++)

{

for (int j = i; j < n; j++)

{

if (mat[j][i] != 0)

{

for (int k = i; k < n; k++)

{

swap(mat[i][k], mat[j][k]);

}

if (i != j)

{

res = (-res + MOD) % MOD;

}

break;

}

}

if (mat[i][i] == 0)

{

res = -1; // 不存在(也就是行列式值为0)

break;

}

for (int j = i + 1; j < n; j++)

{

//int mut = (mat[j][i]\*INV[mat[i][i]])%MOD;//打表逆元

int mut = (mat[j][i] \* inv(mat[i][i], MOD)) % MOD;

for (int k = i; k < n; k++)

{

mat[j][k] = (mat[j][k] - (mat[i][k] \* mut) % MOD + MOD) % MOD;

}

}

res = (res \* mat[i][i]) % MOD;

}

return res;

}

};

int main()

{

Matrix ret;

ret.init();

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i != j && g[i][j])

{

ret.mat[i][j] = -1;

ret.mat[i][i]++;

}

}

}

printf("%d\n", ret.det(n - 1));

return 0;

}

##### 计算生成树个数,不取模

const double eps = 1e-8;

const int MAXN = 110;

int sgn(double x)

{

if (fabs(x) < eps)

{

return 0;

}

if (x < 0)

{

return -1;

}

else

{

return 1;

}

}

double b[MAXN][MAXN];

double det(double a[][MAXN], int n)

{

int i, j, k, sign = 0;

double ret = 1;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

b[i][j] = a[i][j];

}

}

for (i = 0; i < n; i++)

{

if (sgn(b[i][i]) == 0)

{

for (j = i + 1; j < n; j++)

{

if (sgn(b[j][i]) != 0)

{

break;

}

}

if (j == n)

{

return 0;

}

for (k = i; k < n; k++)

{

swap(b[i][k], b[j][k]);

}

sign++;

}

ret \*= b[i][i];

for (k = i + 1; k < n; k++)

{

b[i][k] /= b[i][i];

}

for (j = i+1; j < n; j++)

{

for (k = i+1; k < n; k++)

{

b[j][k] -= b[j][i] \* b[i][k];

}

}

}

if (sign & 1)

{

ret = -ret;

}

return ret;

}

double a[MAXN][MAXN];

int g[MAXN][MAXN];

int main()

{

int T;

int n, m;

int u, v;

scanf("%d", &T);

while (T--)

{

scanf("%d%d", &n, &m);

memset(g, 0, sizeof(g));

while (m--)

{

scanf("%d%d", &u, &v);

u--;

v--;

g[u][v] = g[v][u] = 1;

}

memset(a, 0, sizeof(a));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i != j && g[i][j])

{

a[i][i]++;

a[i][j] = -1;

}

}

}

double ans = det(a, n - 1);

printf("%.0lf\n", ans);

}

return 0;

}

### 有向图最小树形图

/\*

\* 有向图最小树形图

\* INIT: eg置为边表；res置为0；cp[i]置为i；

\* CALL: dirTree(root, nv, ne); res是结果

\*/

#define typec int // type of res

const typec V = 1010;

const typec E = 10010;

const typec inf = 0x3f3f3f3f; // max of res

typec res, dis[V];

int to[V], cp[V], tag[V];

struct Edge

{

int u, v;

typec c;

} eg[E];

int iroot(int i)

{

if (cp[i] == i)

{

return i;

}

return cp[i] = iroot(cp[i]);

}

int dirTree(int root, int nv, int ne) // root:树根

{

// vertex:0~n-1

int i, j, k, circle = 0;

memset(tag, -1, sizeof(tag));

memset(to, -1, sizeof(to));

for (i = 0; i < nv; i++)

{

dis[i] = inf;

}

for (j = 0; j < ne; j++)

{

i = iroot(eg[j].u);

k = iroot(eg[j].v);

if (k != i && dis[k] > eg[j].c)

{

dis[k] = eg[j].c;

to[k] = i;

}

}

to[root] = -1;

dis[root] = 0;

tag[root] = root;

for (i = 0; i < nv; i++)

{

if (cp[i] == i && -1 == tag[i])

{

j = i;

for (; j != -1 && tag[j] == -1; j = to[j])

{

tag[j] = i;

if (j == -1)

{

return 0;

}

if (tag[j] == i)

{

circle = 1;

tag[j] = -2;

for (k = to[j]; k != j; k = to[k])

{

tag[k] = -2;

}

}

}

}

}

if (circle)

{

for (j = 0; j < ne; j++)

{

i = iroot(eg[j].u);

k = iroot(eg[j].v);

if (k != i && tag[k] == -2)

{

eg[j].c -= dis[k];

}

}

for (i = 0; i < nv; i++)

{

if (tag[i] == -2)

{

res += dis[i];

tag[i] = 0;

for (j = to[i]; j != i; j = to[j])

{

res += dis[j];

cp[j] = i;

tag[j] = 0;

}

}

}

if (0 == dirTree(root, nv, ne))

{

return 0;

}

}

else

{

for (i = 0; i < nv; i++)

{

if (cp[i] == i)

{

res += dis[i];

}

}

}

return 1; // 若返回0代表原图不连通

}

### 有向图的强连通分量

##### Tarjan

/\*

\* Tarjan算法

\* 复杂度O(N+M)

\*/

const int MAXN = 20010; // 点数

const int MAXM = 50010; // 边数

struct Edge

{

int to, next;

}edge[MAXM];

int head[MAXN], tot;

int Low[MAXN], DFN[MAXN], Stack[MAXN], Belong[MAXN]; // Belong数组的值是1~scc

int Index, top;

int scc; // 强连通分量的个数

bool Instack[MAXN];

int num[MAXN]; // 各个强连通分量包含点的个数,数组编号1~scc

// num数组不一定需要,结合实际情况

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

void Tarjan(int u)

{

int v;

Low[u] = DFN[u] = ++Index;

Stack[top++] = u;

Instack[u] = true;

for (int i = head[u]; i != -1; i = edge[i].next)

{

v = edge[i].to;

if (!DFN[v])

{

Tarjan(v);

if (Low[u] > Low[v])

{

Low[u] = Low[v];

}

}

else if (Instack[v] && Low[u] > DFN[v])

{

Low[u] = DFN[v];

}

}

if (Low[u] == DFN[u])

{

scc++;

do

{

v = Stack[--top];

Instack[v] = false;

Belong[v] = scc; num[scc]++;

}

while (v != u);

}

return ;

}

void solve(int N)

{

memset(DFN, 0, sizeof(DFN));

memset(Instack, false, sizeof(Instack));

memset(num, 0, sizeof(num));

Index = scc = top = 0;

for (int i = 1; i <= N; i++)

{

if (!DFN[i])

{

Tarjan(i);

}

}

return ;

}

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

return ;

}

##### Kosaraju

/\*

\* 复杂度O(N+M)

\*/

const int MAXN = 20010;

const int MAXM = 50010;

struct Edge

{

int to, next;

} edge1[MAXM], edge2[MAXM]; // edge1是原图G,edge2是逆图GT

int head1[MAXN], head2[MAXN];

bool mark1[MAXN], mark2[MAXN];

int tot1, tot2;

int cnt1, cnt2;

int st[MAXN]; // 对原图进行dfs,点的结束时间从小到大排序

int Belong[MAXN]; // 每个点属于哪个连通分量(0~cnt2-1)

int num; // 中间变量,用来数某个连通分量中点的个数

int setNum[MAXN]; // 强连通分量中点的个数,编号0~cnt2-1

void addedge(int u, int v)

{

edge1[tot1].to = v;

edge1[tot1].next = head1[u];

head1[u] = tot1++;

edge2[tot2].to = u;

edge2[tot2].next = head2[v];

head2[v] = tot2++;

return ;

}

void DFS1(int u)

{

mark1[u] = true;

for (int i = head1[u]; i != -1; i = edge1[i].next)

{

if(!mark1[edge1[i].to])

{

DFS1(edge1[i].to);

}

}

st[cnt1++] = u;

return ;

}

void DFS2(int u)

{

mark2[u] = true;

num++;

Belong[u] = cnt2;

for (int i = head2[u]; i != -1; i = edge2[i].next)

{

if(!mark2[edge2[i].to])

{

DFS2(edge2[i].to);

}

}

return ;

}

void solve(int n) // 点的编号从1开始

{

memset(mark1, false, sizeof(mark1));

memset(mark2, false, sizeof(mark2));

cnt1 = cnt2 = 0;

for (int i = 1; i <= n; i++)

{

if (!mark1[i])

{

DFS1(i);

}

}

for (int i = cnt1 - 1; i >= 0; i--)

{

if (!mark2[st[i]])

{

num = 0;

DFS2(st[i]);

setNum[cnt2++] = num;

}

}

return ;

}

### 双连通分支

##### 点双连通分支

去掉桥,其余的连通分支就是边双连通分支了。一个有桥的连通图要变成边双连通图的话,把双连通子图 收缩为一个点,形成一颗树。需要加的边为(leaf+1)/2 (leaf 为叶子结点个数)

给定一个连通的无向图 G,至少要添加几条边,才能使其变为双连通图。

const int MAXN = 5010; // 点数

const int MAXM = 20010; // 边数,因为是无向图,所以这个值要\*2

struct Edge

{

int to, next;

bool cut; // 是否是桥标记

}edge[MAXM];

int head[MAXN], tot;

int Low[MAXN], DFN[MAXN], Stack[MAXN], Belong[MAXN]; //Belong数组的值是1~block

int Index,top;

int block; // 边双连通块数

bool Instack[MAXN];

int bridge; // 桥的数目

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

edge[tot].cut=false;

head[u] = tot++;

return ;

}

void Tarjan(int u, int pre)

{

int v;

Low[u] = DFN[u] = ++Index;

Stack[top++] = u;

Instack[u] = true;

for (int i = head[u]; i != -1; i = edge[i].next)

{

v = edge[i].to;

if (v == pre)

{

continue;

}

if (!DFN[v])

{

Tarjan(v, u);

if (Low[u] > Low[v])

{

Low[u] = Low[v];

}

if (Low[v] > DFN[u])

{

bridge++;

edge[i].cut = true;

edge[i^1].cut = true;

}

}

else if (Instack[v] && Low[u] > DFN[v])

{

Low[u] = DFN[v];

}

}

if (Low[u] == DFN[u])

{

block++;

do

{

v = Stack[--top]; Instack[v] = false;

Belong[v] = block;

}

while (v != u);

}

return ;

}

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

return ;

}

int du[MAXN]; // 缩点后形成树,每个点的度数

void solve(int n)

{

memset(DFN, 0, sizeof(DFN));

memset(Instack, false, sizeof(Instack));

Index = top = block = 0;

Tarjan(1,0);

int ans = 0;

memset(du, 0, sizeof(du));

for (int i = 1; i <= n; i++)

{

for (int j = head[i]; j != -1; j = edge[j].next)

{

if (edge[j].cut)

{

du[Belong[i]]++;

}

}

}

for (int i = 1; i <= block; i++)

{

if(du[i]==1)

{

ans++;

}

}

// 找叶子结点的个数ans,构造边双连通图需要加边(ans+1)/2

printf("%d\n", (ans + 1) / 2);

}

int main()

{

int n, m;

int u, v;

while (scanf("%d%d", &n, &m) == 2)

{

init();

while (m--)

{

scanf("%d%d",&u,&v);

addedge(u,v);

addedge(v,u);

}

solve(n);

}

return 0;

}

##### 边双连通分支

对于点双连通分支,实际上在求割点的过程中就能顺便把每个点双连通分支求出。建立一个栈,存储 当前双连通分支,在搜索图时,每找到一条树枝边或后向边(非横叉边),就把这条边加入栈中。如果遇到某时满足 DFS(u)<=Low(v),说明u是一个割点,同时把边从栈顶一个个取出,直到遇到了边(u,v), 取出的这些边与其关联的点,组成一个点双连通分支。割点可以属于多个点双连通分支,其余点和每条边只属于且属于一个点双连通分支。

奇圈,二分图判断的染色法,求点双连通分支

/\*

\* POJ 2942 Knights of the Round Table

\* 亚瑟王要在圆桌上召开骑士会议,为了不引发骑士之间的冲突,

\* 并且能够让会议的议题有令人满意的结果,每次开会前都必须对出席会议的骑士有如下要求:

\* 1、 相互憎恨的两个骑士不能坐在直接相邻的2个位置;

\* 2、 出席会议的骑士数必须是奇数,这是为了让投票表决议题时都能有结果。

\* 注意:1、所给出的憎恨关系一定是双向的,不存在单向憎恨关系。

\* 2、由于是圆桌会议,则每个出席的骑士身边必定刚好有2个骑士。

\* 即每个骑士的座位两边都必定各有一个骑士。

\* 3、一个骑士无法开会,就是说至少有3个骑士才可能开会。

\* 首先根据给出的互相憎恨的图中得到补图。

\* 然后就相当于找出不能形成奇圈的点。

\* 利用下面两个定理:

\* (1)如果一个双连通分量内的某些顶点在一个奇圈中(即双连通分量含有奇圈), 那么这个双连通分量的其他顶点也在某个奇圈中;

\* (2)如果一个双连通分量含有奇圈,则他必定不是一个二分图。反过来也成立,这是一个充要条件。

\* 所以本题的做法,就是对补图求点双连通分量。然后对于求得的点双连通分量,使用染色法判断是不是二分图,不是二分图,这个双连通分量的点是可以存在的

\*/

const int MAXN = 1010;

const int MAXM = 2000010;

struct Edge

{

int to, next;

} edge[MAXM];

int head[MAXN], tot;

int Low[MAXN], DFN[MAXN], Stack[MAXN], Belong[MAXN];

int Index,top;

int block; // 点双连通分量的个数

bool Instack[MAXN];

bool can[MAXN];

bool ok[MAXN]; // 标记

int tmp[MAXN]; // 暂时存储双连通分量中的点

int cc; // tmp的计数

int color[MAXN];// 染色

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

bool dfs(int u, int col) // 染色判断二分图

{

color[u] = col;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (!ok[v])

{

continue;

}

if (color[v] != -1)

{

if (color[v]==col)

{

return false;

}

continue;

}

if (!dfs(v,!col))

{

return false;

}

}

return true;

}

void Tarjan(int u, int pre)

{

int v;

Low[u] = DFN[u] = ++Index;

Stack[top++] = u;

Instack[u] = true;

for (int i = head[u]; i != -1; i = edge[i].next)

{

v = edge[i].to;

if (v == pre)

{

continue;

}

if (!DFN[v])

{

Tarjan(v, u);

if (Low[u] > Low[v])

{

Low[u] = Low[v];

}

if (Low[v] >= DFN[u])

{

block++;

int vn;

cc = 0;

memset(ok, false, sizeof(ok));

do

{

vn = Stack[--top];

Belong[vn] = block;

Instack[vn] = false;

ok[vn] = true;

tmp[cc++] = vn;

}

while (vn!=v);

ok[u] = 1;

memset(color, -1, sizeof(color));

if (!dfs(u,0))

{

can[u] = true;

while (cc--)

{

can[tmp[cc]] = true;

}

}

}

}

else if (Instack[v] && Low[u] > DFN[v])

{

Low[u] = DFN[v];

}

}

}

void solve(int n)

{

memset(DFN, 0, sizeof(DFN));

memset(Instack, false, sizeof(Instack));

Index = block = top = 0;

memset(can, false, sizeof(can));

for (int i = 1; i <= n; i++)

{

if (!DFN[i])

{

Tarjan(i, -1);

}

}

int ans = n;

for (int i = 1; i <= n; i++)

{

if(can[i])

{

ans--;

}

}

printf("%d\n", ans);

}

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

}

int g[MAXN][MAXN];

int main()

{

int n, m;

int u, v;

while (scanf("%d%d", &n, &m) == 2)

{

if (n == 0 && m == 0)

{

break;

}

init();

memset(g, 0, sizeof(g));

while (m--)

{

scanf("%d%d", &u, &v);

g[u][v] = g[v][u] = 1;

}

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= n; j++)

{

if(i != j && g[i][j] == 0)

{

addedge(i, j);

}

}

}

solve(n);

}

return 0;

}

### 弦图判断

/\*

\* 弦图判断

\* INIT: g[][]置为邻接矩阵;

\* CALL: mcs(n); peo(n);

\* 第一步: 给节点编号 mcs(n)

\* 设已编号的节点集合为A, 未编号的节点集合为B

\* 开始时A为空, B包含所有节点.

\* for num=n-1 downto 0 do {

\* 在B中找节点x, 使与x相邻的在A集合中的节点数最多,

\* 将x编号为num，并从B移入A。

\* }

\* 第二部：检查peo(n)

\* for num=0 to n-1 do {

\* 对编号为num的点x，设所有编号>num且与x相邻的点集为C

\* 在C中找出编号最小的节点y，

\* 若C中存在x != y，使得y与x之间无边，则此图不是弦图。

\* }

\* 检查完毕, 则此图是弦图.

\*/

const int V = 10010;

int g[V][V], order[V], inv[V], tag[V];

void mcs(int n)

{

int i, j, k;

memset(tag, 0, sizeof(tag));

memset(order, -1, sizeof(order));

for (i = n - 1; i >= 0; i--)

{ // vertex:0~n-1

for (j = 0; order[j] >= 0; j++);

for (k = j + 1; k < n; k++)

{

if (order[k] < 0 && tag[k] > tag[j])

{

j = k;

}

}

order[j] = i, inv[i] = j;

for (k = 0; k < n; k++)

{

if (g[j][k])

{

tag[k]++;

}

}

}

return ;

}

int peo(int n)

{

int i, j, k, w, min;

for (i = n - 2; i >= 0; i--)

{

j = inv[i], w = -1, min = n;

for (k = 0; k < n; k++)

{

if (g[j][k] && order[k] > order[j] && order[k] < min)

{

min = order[k], w=k;

}

}

if (w < 0)

{

continue;

}

for (k = 0; k < n; k++)

{

if (g[j][k] && order[k] > order[w] && !g[k][w])

{

return 0; // no

}

}

}

return 1; // yes

}

### 弦图的PERFECT ELIMINATION点排列

/\*

\* INIT: g[][]置为邻接矩阵;

\* CALL: cardinality(n);tag[i]为排列中第i个点的标号;

\*/

const int V = 10010;

int tag[V], g[V][V], deg[V], vis[V];

void cardinality(int n)

{

int i, j, k;

memset(deg, 0, sizeof(deg));

memset(vis, 0, sizeof(vis));

for (i = n - 1; i >= 0; i--)

{

for (j = 0, k = -1; j < n; j++)

{

if (0 == vis[j])

{

if (k == -1 || deg[j] > deg[k])

{

k = j;

}

}

}

vis[k] = 1, tag[i] = k;

for (j = 0; j<n; j++)

{

if (0 == vis[j] && g[k][j])

{

deg[j]++;

}

}

}

return ;

}

### 稳定婚姻问题

/\*

\* 稳定婚姻问题O(n^2)

\*/

const int N = 1001;

struct People

{

bool state;

int opp, tag;

int list[N]; // man使用

int priority[N]; // woman使用，有必要的话可以和list合并，以节省空间

void Init()

{

state = tag = 0;

}

} man[N], woman[N];

struct R

{

int opp;

int own;

} requst[N];

int n;

void Input();

void Output();

void stableMatching();

int main()

{

Input();

stableMatching();

Output();

return 0;

}

void Input()

{

scanf("%d\n", &n);

int i, j, ch;

for (i = 0; i < n; ++i)

{

man[i].Init();

for(j = 0; j < n; ++j)

{ // 按照man的意愿递减排序

scanf("%d", &ch);

man[i].list[j] = ch - 1;

}

}

for (i = 0; i < n; ++i)

{

woman[i].Init();

for (j = 0; j < n; ++j)

{ // 按照woman的意愿递减排序,但是，存储方法与man不同

scanf("%d", &ch);

woman[i].priority[ch - 1] = j;

}

}

return ;

}

void stableMatching()

{

int k;

for (k = 0; k < n; ++k)

{

int i, id = 0;

for (i = 0; i < n; ++i)

{

if (man[i].state == 0)

{

requst[id].opp = man[i].list[man[i].tag];

requst[id].own = i;

man[i].tag += 1;

++id;

}

}

if (id == 0)

{

break;

}

for (i = 0; i < id; i++)

{

if (woman[requst[i].opp].state == 0)

{

woman[requst[i].opp].opp = requst[i].opp;

woman[requst[i].opp].state = 1;

man[requst[i].own].state = 1;

man[requst[i].own].opp = requst[i].opp;

}

else

{

if (woman[requst[i].opp].priority[woman[requst[i].opp].opp] >woman[requst[i].opp].priority[requst[i].own])

{

man[woman[requst[i].opp].opp].state = 0;

woman[requst[i].opp].opp = requst[i].own;

man[requst[i].own].state = 1;

man[requst[i].own].opp = requst[i].opp;

}

}

}

}

return ;

}

void Output()

{

for (int i = 0; i < n; i++)

{

printf("%d\n", man[i].opp + 1);

}

return ;

}

### 拓扑排序

/\*

\* 拓扑排序

\* INIT:edge[][]置为图的邻接矩阵;cnt[0...i...n-1]:顶点i的入度。

\*/

const int MAXV = 1010;

int edge[MAXV][MAXV];

int cnt[MAXV];

void TopoOrder(int n)

{

int i, top = -1;

for (i = 0; i < n; ++i)

{

if (cnt[i] == 0)

{ // 下标模拟堆栈

cnt[i] = top;

top = i;

}

}

for (i = 0; i < n; i++)

{

if (top == -1)

{

printf("存在回路\n");

return ;

}

else

{

int j = top;

top = cnt[top];

printf("%d", j);

for (int k = 0; k < n; k++)

{

if (edge[j][k] && (--cnt[k]) == 0)

{

cnt[k] = top;

top = k;

}

}

}

}

}

### 无向图连通分支

/\*

\* 无向图连通分支(dfs/bfs邻接阵)

\* DFS / BFS / 并查集

\* 返回分支数,id返回1.分支数的值

\* 传入图的大小n和邻接阵mat,不相邻点边权0

\*/

#define MAXN 100

void search(int n, int mat[][MAXN], int\* dfn, int\* low, int now, int& cnt, int& tag, int\* id, int\* st, int& sp)

{

int i, j;

dfn[st[sp++]=now] = low[now] = ++cnt;

for (i = 0; i < n; i++)

{

if (mat[now][i])

{

if (!dfn[i])

{

search(n, mat, dfn, low, i, cnt, tag, id, st, sp);

if (low[i] < low[now])

{

low[now]=low[i];

}

}

else if (dfn[i] < dfn[now])

{

for (j = 0; j < sp && st[j] != i; j++)

{

if (j < cnt && dfn[i] < low[now])

{

low[now] = dfn[i];

}

}

}

}

}

if (low[now] == dfn[now])

{

for (tag++; st[sp] != now; id[st[--sp]] = tag);

}

}

int find\_components(int n, int mat[][MAXN], int\* id)

{

int ret = 0, i, cnt, sp, st[MAXN], dfn[MAXN], low[MAXN];

for (i = 0; i < n; dfn[i++] = 0);

for (sp = cnt = i = 0; i < n; i++)

{

if (!dfn[i])

{

search(n, mat, dfn, low, i, cnt, ret, id, st, sp);

}

}

return ret;

}

### 有向图强连通分支

/\*

\* 有向图强连通分支(dfs/bfs邻接阵)O(n^2)

\* 返回分支数,id返回1..分支数的值

\* 传入图的大小n和邻接阵mat,不相邻点边权0

\*/

#define MAXN 100

int find\_components(int n, int mat[][MAXN], int\* id)

{

int ret = 0, a[MAXN], b[MAXN], c[MAXN], d[MAXN], i, j, k, t;

for (k = 0; k < n; id[k++] = 0);

for (k = 0; k < n; k++)

{

if (!id[k])

{

for (i = 0; i < n; i++)

{

a[i] = b[i] = c[i] = d[i] = 0;

}

a[k] = b[k] = 1;

for (t = 1; t;)

{

for (t = i = 0; i < n; i++)

{

if (a[i] && !c[i])

{

for (c[i] = t = 1, j = 0; j < n; j++)

{

if (mat[i][j] && !a[j])

{

a[j] = 1;

}

}

}

if (b[i] && !d[i])

{

for (d[i] = t = 1, j = 0; j < n; j++)

{

if (mat[j][i] && !b[j])

{

b[j] = 1;

}

}

}

}

}

for (ret++, i = 0; i < n; i++)

{

if (a[i] & b[i])

{

id[i] = ret;

}

}

}

}

return ret;

}

### 有向图最小点基

/\*

\* 有向图最小点基(邻接阵)O(n^2)

\* 点基B满足:对于任意一个顶点Vj,一定存在B中的一个Vi,使得Vi是Vj的前代。

\* 返回点基大小和点基 传入图的大小n和邻接阵mat,不相邻点边权0 需要调用强连通分支

\* find\_components(n, mat, id);参考《有向图强连通分支》

\*/

#define MAXN 100

int base\_vertex(int n, int mat[][MAXN], int\* sets)

{

int ret=0, id[MAXN], v[MAXN], i, j;

j = find\_components(n, mat, id);

for (i = 0; i < j; v[i++] = 1);

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (id[i] != id[j] && mat[i][j])

{

v[id[j] - 1] = 0;

}

}

}

for (i = 0; i < n; i++)

{

if (v[id[i] - 1])

{

v[id[sets[ret++] = i] - 1] = 0;

}

}

return ret;

}

### Floyd求最小环

令e(u, v)表示u和v之间的连边，令min(u, v)表示删除u和v之间的连边之后u和v之间的最短路, 最小环则是min(u, v) + e(u, v). 时间复杂度是 O(EV^2).   
改进算法   
在floyd的同时,顺便算出最小环   
g[i][j]=i, j之间的边长

dist:=g;

for k:=1 to n do

begin

for i:=1 to k-1 do

for j:=i+1 to k-1 do

answer:=min(answer, dist[i][j]+g[i][k]+g[k][j]);

for i:=1 to n do

for j:=1 to n do

dist[i][j]:=min(dist[i][j], dist[i][k]+dist[k][j]);

end;

最小环改进算法的证明

一个环中的最大结点为k(编号最大), 与他相连的两个点为i, j, 这个环的最短长度为g[i][k]+g[k][j]+i到j的路径中所有结点编号都小于k的最短路径长度. 根据floyd的原理, 在最外层循环做了k-1次之后, dist[i][j]则代表了i到j的路径中所有结点编号都小于k的最短路径综上所述,该算法一定能找到图中最小环。

const int INF = 0x3f3f3f3f;

const int MAXN = 110;

int n, m; // n:节点个数, m:边的个数

int g[MAXN][MAXN]; // 无向图

int dist[MAXN][MAXN]; // 最短路径

int r[MAXN][MAXN]; // r[i][j]: i到j的最短路径的第一步

int out[MAXN], ct; // 记录最小环

int solve(int i, int j, int k)

{ // 记录最小环

ct = 0;

while (j != i)

{

out[ct++] = j;

j = r[i][j];

}

out[ct++] = i;

out[ct++] = k;

return 0;

}

int main()

{

while (scanf("%d%d", &n, &m) != EOF)

{

int i, j, k;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

g[i][j] = INF;

r[i][j] = i;

}

}

for (i = 0; i < m; i++)

{

int x, y, l;

scanf("%d%d%d", &x, &y, &l);

--x;

--y;

if (l < g[x][y])

{

g[x][y] = g[y][x] = l;

}

}

memmove(dist, g, sizeof(dist));

int Min = INF; // 最小环

for (k = 0; k < n; k++)

{ // Floyd

for (i = 0; i < k; i++) // 一个环中的最大结点为k(编号最大)

{

if (g[k][i] < INF)

{

for (j = i + 1; j < k; j++)

{

if (dist[i][j] < INF && g[k][j] < INF && Min > dist[i][j] + g[k][i] + g[k][j])

{

Min = dist[i][j] + g[k][i] + g[k][j];

solve(i, j, k); // 记录最小环

}

}

}

}

for (i = 0; i < n; i++)

{

if (dist[i][k] < INF)

{

for (j = 0; j < n; j++)

{

if (dist[k][j] < INF && dist[i][j] > dist[i][k]+dist[k][j])

{

dist[i][j] = dist[i][k] + dist[k][j];

r[i][j] = r[k][j];

}

}

}

}

}

if (Min < INF)

{

for (ct--; ct >= 0; ct--)

{

printf("%d", out[ct] + 1);

if (ct)

{

printf(" ");

}

}

}

else

{

printf("No solution.");

}

printf("\n");

}

return 0;

}

### 2-SAT问题

/\*

\* 2-sat 问题

\* N个集团,每个集团2个人,现在要想选出尽量多的人,

\* 且每个集团只能选出一个人。如果两人有矛盾,他们不能同时被选中

\* 问最多能选出多少人

\*/

const int MAXN = 3010;

int n, m;

int g[3010][3010], ct[3010], f[3010];

int x[3010], y[3010];

int prev[MAXN], low[MAXN], stk[MAXN], sc[MAXN];

int cnt[MAXN];

int cnt0, ptr, cnt1;

void dfs(int w)

{

int min(0);

prev[w] = cnt0++;

low[w] = prev[w];

min = low[w];

stk[ptr++] = w;

for (int i = 0; i < ct[w]; ++i)

{

int t = g[w][i];

if (prev[t] == -1)

{

dfs(t);

}

if (low[t] < min)

{

min = low[t];

}

}

if (min < low[w])

{

low[w] = min;

return ;

}

do

{

int v = stk[--ptr];

sc[v] = cnt1;

low[v] = MAXN;

} while(stk[ptr] != w);

++cnt1;

return ;

}

void Tarjan(int N)

{ // 传入N为点数,结果保存在sc数组中,同一标号的点在同一个强连通分量内,

// 强连通分量数为cnt1

cnt0 = cnt1 = ptr = 0;

int i;

for (i = 0; i < N; ++i)

{

prev[i] = low[i] = -1;

}

for (i = 0; i < N; ++i)

{

if (prev[i] == -1)

{

dfs(i);

}

}

return ;

}

int solve()

{

Tarjan(n);

for (int i = 0; i < n; i++)

{

if (sc[i] == sc[f[i]])

{

return 0;

}

}

return 1;

}

int check(int Mid)

{

for (int i = 0; i < n; i++)

{

ct[i] = 0;

}

for (int i = 0; i < Mid; i++)

{

g[f[x[i]]][ct[f[x[i]]]++] = y[i];

g[f[y[i]]][ct[f[y[i]]]++] = x[i];

}

return solve();

}

int main()

{

while (scanf("%d%d", &n, &m) != EOF && n + m)

{

for (int i = 0; i < n; i++)

{

int p, q;

scanf("%d%d", &p, &q);

f[p] = q, f[q] = p;

}

for (int i = 0; i < m; i++)

{

scanf("%d%d", &x[i], &y[i]);

}

n \*= 2;

int Min = 0, Max = m + 1;

while (Min + 1 < Max)

{

int Mid = (Min + Max) / 2;

if (check(Mid))

{

Min = Mid;

}

else

{

Max = Mid;

}

}

printf("%d\n", Min);

}

return 0;

}

### 树的重心

typedef long long ll;

typedef pair<int, int> pll;

const int INF = 0x3f3f3f3f;

const int MAXN = 100000 + 10;

int n;

/\* 树的重心

\* 初始化 vis[] son[] 为 0

\* 初始化 sz 为 INF

\*/

int zx, sz;

int son[MAXN], vis[MAXN];

vector<pll> edge[MAXN];

void init()

{

for (int i = 1; i <= n; i++)

{

edge[i].clear();

}

memset(vis, 0, sizeof(vis));

sz = INF;

zx = -1;

}

void dfs(int r)

{

vis[r] = 1;

son[r] = 0;

int tmp = 0;

for (int i = 0; i < edge[r].size(); i++)

{

int v = edge[r][i].second;

if (!vis[v])

{

dfs(v);

son[r] += son[v] + 1;

tmp = max(tmp, son[v] + 1);

}

}

tmp = max(tmp, n - son[r] - 1);

if (tmp < sz)

{

zx = r;

sz = tmp;

}

}

### 最小生成树-Kruskal

void Kruskal() {

ans = 0;

for (int i = 0; i<len; i++) {

if (Find(edge[i].a) != Find(edge[i].b)) {

Union(edge[i].a, edge[i].b);

ans += edge[i].len;

}

}

}

#include<iostream>

#include<cstring>

#include<cstdio>

#include<algorithm>

using namespace std;

const int maxn=10000+5;

int pre[maxn];

int n,len;

struct node{

int u,v,w;

}stu[maxn];

bool cmp(node x,node y)

{

return x.w<y.w;

}

int find\_pre(int x)

{

if(x!=pre[x])

pre[x]=find\_pre(pre[x]);

return pre[x];

}

int join(int a,int b)

{

int x=find\_pre(a);

int y=find\_pre(b);

if(x!=y)

{

pre[x]=y;

return 1;

}

return 0;

}

void Kruskal(){

int cnt=0,sum=0;

for(int i=1;i<=n;i++)

pre[i]=i;

for(int i=0;i<len;i++){

if(join(stu[i].u,stu[i].v)){

++cnt;

sum+=stu[i].w;

}

if(cnt==n-1)

break;

}

cout<<sum<<endl;

}

int main(){

char s,s2;

int w,m;

while(cin>>n&&n){

len=0;

memset(pre,0,sizeof(pre));

for(int i=1;i<=n-1;i++)

{

getchar();

scanf("%c ",&s);

cin>>m;

for(int j=0;j<m;j++){

stu[len].u=s-'A';

scanf(" %c %d",&s2,&w);

stu[len].v=s2-'A';

stu[len++].w=w;

}

}

sort(stu,stu+len,cmp);

Kruskal();

}

return 0;

}

### 最小生成树-Prim

/\*

|Prim算法|

|适用于 稠密图 求最小生成树|

|堆优化版，时间复杂度：O(elgn)|

\*/

struct node {

int v, len;

node(int v = 0, int len = 0) :v(v), len(len) {}

bool operator < (const node &a)const { // 加入队列的元素自动按距离从小到大排序

return len> a.len;

}

};

vector<node> G[maxn];

int vis[maxn];

int dis[maxn];

void init() {

for (int i = 0; i<maxn; i++) {

G[i].clear();

dis[i] = INF;

vis[i] = false;

}

}

int Prim(int s) {

priority\_queue<node>Q; // 定义优先队列

int ans = 0;

Q.push(node(s,0)); // 起点加入队列

while (!Q.empty()) {

node now = Q.top(); Q.pop(); // 取出距离最小的点

int v = now.v;

if (vis[v]) continue; // 同一个节点，可能会推入2次或2次以上队列，这样第一个被标记后，剩下的需要直接跳过。

vis[v] = true; // 标记一下

ans += now.len;

for (int i = 0; i<G[v].size(); i++) { // 开始更新

int v2 = G[v][i].v;

int len = G[v][i].len;

if (!vis[v2] && dis[v2] > len) {

dis[v2] = len;

Q.push(node(v2, dis[v2])); // 更新的点加入队列并排序

}

}

}

return ans;

}

### 单源最短路径-Dijkstra

/\*

|Dijkstra算法|

|适用于边权为正的有向图或者无向图|

|求从单个源点出发，到所有节点的最短路|

|优化版：时间复杂度 O(elbn)|

\*/

struct node {

int v, len;

node(int v = 0, int len = 0) :v(v), len(len) {}

bool operator < (const node &a)const { // 距离从小到大排序

return len > a.len;

}

};

vector<node>G[maxn];

bool vis[maxn];

int dis[maxn];

void init() {

for (int i = 0; i<maxn; i++) {

G[i].clear();

vis[i] = false;

dis[i] = INF;

}

}

int dijkstra(int s, int e) {

priority\_queue<node>Q;

Q.push(node(s, 0)); // 加入队列并排序

dis[s] = 0;

while (!Q.empty()) {

node now = Q.top(); // 取出当前最小的

Q.pop();

int v = now.v;

if (vis[v]) continue; // 如果标记过了, 直接continue

vis[v] = true;

for (int i = 0; i<G[v].size(); i++) { // 更新

int v2 = G[v][i].v;

int len = G[v][i].len;

if (!vis[v2] && dis[v2] > dis[v] + len) {

dis[v2] = dis[v] + len;

Q.push(node(v2, dis[v2]));

}

}

}

return dis[e];

}

### 最短路径快速算法-SPFA

/\*

|Dijkstra算法|

|适用于边权为正的有向图或者无向图|

|求从单个源点出发，到所有节点的最短路|

|优化版：时间复杂度 O(elbn)|

\*/

struct node {

int v, len;

node(int v = 0, int len = 0) :v(v), len(len) {}

bool operator < (const node &a)const { // 距离从小到大排序

return len > a.len;

}

};

vector<node>G[maxn];

bool vis[maxn];

int dis[maxn];

void init() {

for (int i = 0; i<maxn; i++) {

G[i].clear();

vis[i] = false;

dis[i] = INF;

}

}

int dijkstra(int s, int e) {

priority\_queue<node>Q;

Q.push(node(s, 0)); // 加入队列并排序

dis[s] = 0;

while (!Q.empty()) {

node now = Q.top(); // 取出当前最小的

Q.pop();

int v = now.v;

if (vis[v]) continue; // 如果标记过了, 直接continue

vis[v] = true;

for (int i = 0; i<G[v].size(); i++) { // 更新

int v2 = G[v][i].v;

int len = G[v][i].len;

if (!vis[v2] && dis[v2] > dis[v] + len) {

dis[v2] = dis[v] + len;

Q.push(node(v2, dis[v2]));

}

}

}

return dis[e];

}

### 弗洛伊德算法-Floyd-Warshall

/\*

|Floyd算法|

|任意点对最短路算法|

|求图中任意两点的最短距离的算法|

\*/

for (int i = 0; i < n; i++) { // 初始化为0

for (int j = 0; j < n; j++)

scanf("%lf", &dis[i][j]);

}

for (int k = 0; k < n; k++) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);

}

}

}

# Network 网络流

### 二分图匹配相关

##### 匈牙利算法（邻接矩阵+DFS）

/\*

\* 初始化:g[][]两边顶点的划分情况

\* 建立g[i][j]表示i->j的有向边就可以了,是左边向右边的匹配

\* g没有边相连则初始化为0

\* uN是匹配左边的顶点数,vN是匹配右边的顶点数

\* 调用:res=hungary();输出最大匹配数

\* 优点:适用于稠密图,DFS找增广路,实现简洁易于理解

\* 时间复杂度:O(VE)

\*/

//顶点编号从0开始的

const int MAXN = 510;

int uN, vN; // u,v的数目,使用前面必须赋值

int g[MAXN][MAXN]; // 邻接矩阵

int linker[MAXN];

bool used[MAXN];

bool dfs(int u)

{

for (int v = 0; v < vN; v++)

{

if (g[u][v] && !used[v])

{

used[v] = true;

if (linker[v] == -1 || dfs(linker[v]))

{

linker[v] = u;

return true;

}

}

}

return false;

}

int hungary()

{

int res = 0;

memset(linker, -1, sizeof(linker));

for (int u = 0; u < uN; u++)

{

memset(used, false, sizeof(used));

if (dfs(u))

{

res++;

}

}

return res;

}

##### 匈牙利算法（邻接表+DFS）

/\*

\* 使用前用init()进行初始化,给uN赋值

\* 加边使用函数addedge(u,v)

\*/

const int MAXN = 5010; // 点数的最大值

const int MAXM = 50010; // 边数的最大值

struct Edge

{

int to, next;

} edge[MAXM];

int head[MAXN], tot;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

return ;

}

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

int linker[MAXN];

bool used[MAXN];

int uN;

bool dfs(int u)

{

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (!used[v])

{

used[v] = true;

if (linker[v] == -1 || dfs(linker[v]))

{

linker[v] = u;

return true;

}

}

}

return false;

}

int hungary()

{

int res = 0;

memset(linker, -1, sizeof(linker));

for (int u = 0; u < uN; u++) // 点的编号0~uN-1

{

memset(used, false, sizeof(used));

if (dfs(u))

{

res++;

}

}

return res;

}

##### 匈牙利算法（邻接矩阵+BFS）

/\*

\* INIT: g[][]邻接矩阵;

\* CALL: res = MaxMatch();Nx, Ny初始化!!!

\* 优点:适用于稀疏二分图,边较少,增广路较短。

\* 匈牙利算法的理论复杂度是O(VE)

\*/

const int MAXN = 1000;

int g[MAXN][MAXN], Mx[MAXN], My[MAXN], Nx, Ny;

int chk[MAXN], Q[MAXN], prev[MAXN];

int MaxMatch()

{

int res = 0;

int qs, qe;

memset(Mx, -1, sizeof(Mx));

memset(My, -1, sizeof(My));

memset(chk, -1, sizeof(chk));

for (int i = 0; i < Nx; i++)

{

if (Mx[i] == -1)

{

qs = qe = 0;

Q[qe++] = i;

prev[i] = -1;

bool flag = 0;

while (qs < qe && !flag)

{

int u = Q[qs];

for (int v = 0; v < Ny && !flag; v++)

{

if (g[u][v] && chk[v] != i)

{

chk[v] = i; Q[qe++] = My[v];

if (My[v] >= 0)

{

prev[My[v]] = u;

}

else

{

flag = 1;

int d = u, e = v;

while (d != -1)

{

int t = Mx[d];

Mx[d] = e;

My[e] = d;

d = prev[d];

e = t;

}

}

}

}

qs++;

}

if (Mx[i] != -1)

{

res++;

}

}

}

return res;

}

##### Hopcroft-Carp算法(邻接矩阵+DFS)

/\*

\* INIT: g[][]邻接矩阵；

\* CALL: res = MaxMatch(); Nx, Ny要初始化！！！

\* 时间复杂度: O(V^0.5 \* E)

\*/

const int MAXN = 3001;

const int INF = 1 << 28;

int g[MAXN][MAXN], Mx[MAXN], My[MAXN], Nx, Ny;

int dx[MAXN], dy[MAXN], dis;

bool vst[MAXN];

bool searchP()

{

queue<int> Q;

dis = INF;

memset(dx, -1, sizeof(dx));

memset(dy, -1, sizeof(dy));

for (int i = 0; i < Nx; i++)

{

if (Mx[i] == -1)

{

Q.push(i); dx[i] = 0;

}

}

while (!Q.empty())

{

int u = Q.front();

Q.pop();

if (dx[u] > dis)

{

break;

}

for (int v = 0; v < Ny; v++)

{

if (g[u][v] && dy[v] == -1)

{

dy[v] = dx[u]+1;

if (My[v] == -1)

{

dis = dy[v];

}

else

{

dx[My[v]] = dy[v] + 1;

Q.push(My[v]);

}

}

}

}

return dis != INF;

}

bool DFS(int u)

{

for (int v = 0; v < Ny; v++)

{

if (!vst[v] && g[u][v] && dy[v] == dx[u] + 1)

{

vst[v] = 1;

if (My[v] != -1 && dy[v] == dis)

{

continue;

}

if (My[v] == -1 || DFS(My[v]))

{

My[v] = u; Mx[u] = v;

return 1;

}

}

}

return 0;

}

int MaxMatch()

{

int res = 0;

memset(Mx, -1, sizeof(Mx));

memset(My, -1, sizeof(My));

while (searchP())

{

memset(vst, 0, sizeof(vst));

for (int i = 0; i < Nx; i++)

{

if (Mx[i] == -1 && DFS(i))

{

res++;

}

}

}

return res;

}

##### Hopcroft-Carp算法(邻接表+DFS)

/\*

\* 复杂度O(sqrt(n)\*E)

\* 邻接表存图,vector实现

\* vector先初始化,然后假如边

\* uN为左端的顶点数,使用前赋值(点编号0开始)

\*/

const int MAXN = 3000;

const int INF = 0x3f3f3f3f;

vector<int>G[MAXN];

int uN;

int Mx[MAXN], My[MAXN];

int dx[MAXN], dy[MAXN];

int dis;

bool used[MAXN];

bool SearchP()

{

queue<int>Q;

dis = INF;

memset(dx, -1, sizeof(dx));

memset(dy, -1, sizeof(dy));

for (int i = 0 ; i < uN; i++)

{

if(Mx[i] == -1)

{

Q.push(i);

dx[i] = 0;

}

}

while (!Q.empty())

{

int u = Q.front();

Q.pop();

if (dx[u] > dis)

{

break;

}

int sz = (int)G[u].size();

for (int i = 0; i < sz; i++)

{

int v = G[u][i];

if (dy[v] == -1)

{

dy[v] = dx[u] + 1;

if (My[v] == -1)

{

dis = dy[v];

}

else

{

dx[My[v]] = dy[v] + 1;

Q.push(My[v]);

}

}

}

}

return dis != INF;

}

bool DFS(int u)

{

int sz = (int)G[u].size();

for (int i = 0; i < sz; i++)

{

int v = G[u][i];

if (!used[v] && dy[v] == dx[u] + 1)

{

used[v] = true;

if (My[v] != -1 && dy[v] == dis)

{

continue;

}

if (My[v] == -1 || DFS(My[v]))

{

My[v] = u;

Mx[u] = v;

return true;

}

}

}

return false;

}

int MaxMatch()

{

int res = 0;

memset(Mx, -1, sizeof(Mx));

memset(My, -1, sizeof(My));

while (SearchP())

{

memset(used, false, sizeof(used));

for (int i = 0; i < uN; i++)

{

if(Mx[i] == -1 && DFS(i))

{

res++;

}

}

}

return res;

}

##### 二分图最佳匹配

Kuhn Munkras算法

/\*

\* 邻接距阵形式,复杂度O(m\*m\*n) 返回最佳匹配值,传入二分图大小m,n

\* 邻接距阵mat,表示权,match1,match2返回一个最佳匹配,未匹配顶点

\* match值为-1,一定注意m<=n,否则循环无法终止,最小权匹配可将权值

\* 取相反数

\* 初始化:for (i = 0; i < MAXN; ++i)

\* for (j = 0; j < MAXN ; ++j)

\* mat[i][j] = -inf;

\* 对于存在的边:mat[i][j] = val ; // 注意,不能有负值

\*/

#define MAXN 310

#define inf 1000000000

#define \_clr(x) memset(x, -1, sizeof(int) \* MAXN)

int kuhn\_munkras(int m, int n, int mat[][MAXN], int \*match\_1, int \*match\_2)

{

int s[MAXN], t[MAXN], l\_1[MAXN], l\_2[MAXN];

int p, q, ret = 0;

int i, j, k;

for (i = 0; i < m; i++)

{

for (l\_1[i] = -inf, j = 0; j < n; j++)

{

l\_1[i] = mat[i][j] > l\_1[i] ? mat[i][j] : l\_1[i];

}

if (l\_1[i] == -inf)

{

return -1; // 无结果

}

}

for (i = 0; i < n; l\_2[i++] = 0);

for (\_clr(match\_1), \_clr(match\_2), i = 0; i < m; i++)

{

for (\_clr(t), s[p = q = 0] = i; p <= q && match\_1[i] < 0; p++)

{

for (k = s[p], j = 0; j < n && match\_1[i] < 0; p++)

{

if (l\_1[k] + l\_2[j] == mat[k][j] && t[j] < 0)

{

s[++q] = match\_2[j], t[j] = k;

if (s[q] < 0)

{

for (p = j; p >= 0; j = p)

{

match\_2[j] = k = t[j];

p = match\_1[k];

match\_1[k] = j;

}

}

}

}

}

if (match\_1[i] < 0)

{

for (i--, p = inf, k = 0; k <= q; k++)

{

for (j = 0; j < n; j++)

{

if (t[j] < 0 && l\_1[s[k]] + l\_2[j] - mat[s[k]][j] < p)

{

p = l\_1[s[k]] + l\_2[j] - mat[s[k]][j];

}

}

}

for (j = 0; j < n; l\_2[j] += t[j] < 0 ? 0 : p, j++);

for (k = 0; k <= q; l\_1[s[k++]] -= p);

}

}

for (i = 0; i < m; i++)

{ // if处理无匹配的情况!!

if (match\_1[i] < 0) // ???

{

return -1;

}

if (mat[i][match\_1[i]] <= -inf) // ???

{

return -1;

}

ret += mat[i][match\_1[i]];

}

return ret;

}

##### 二分图多重匹配

const int MAXN = 1010;

const int MAXM = 510;

int uN, vN;

int g[MAXN][MAXM];

int linker[MAXM][MAXN];

bool used[MAXM];

int num[MAXM]; // 右边最大的匹配数

bool dfs(int u)

{

for (int v = 0; v < vN; v++)

{

if (g[u][v] && !used[v])

{

used[v] = true;

if (linker[v][0] < num[v])

{

linker[v][++linker[v][0]] = u;

return true;

}

for (int i = 1; i <= num[0]; i++)

{

if (dfs(linker[v][i]))

{

linker[v][i] = u;

return true;

}

}

}

}

return false;

}

int hungary()

{

int res = 0;

for (int i = 0; i < vN; i++)

{

linker[i][0] = 0;

}

for (int u = 0; u < uN; u++)

{

memset(used, false, sizeof(used));

if (dfs(u))

{

res++;

}

}

return res;

}

### 无向图最小割

/\*

\* INIT: 初始化邻接矩阵g[][]

\* CALL: res = mincut(n);

\* 注: Stoer-Wagner Minimum Cut;

\* 找边的最小集合，若其被删去则图变得不连通（我们把这种形式称为最小割问题）

\*/

#define typec int // type of res

const typec inf = 0x3f3f3f3f; // max of res

const typec maxw = 1000; // maximum edge weight

const typec V = 10010;

typec g[V][V], w[V];

int a[V], v[V], na[V];

typec minCut(int n)

{

int i, j, pv, zj;

typec best = maxw \* n \* n;

for (i = 0; i < n; i++)

{

v[i] = i; // vertex: 0 ~ n-1

}

while (n > 1)

{

for (a[v[0]] = 1, i = 1; i < n; i++)

{

a[v[i]] = 0;

na[i - 1] = i;

w[i] = g[v[0]][v[i]];

}

for (pv = v[0], i = 1; i < n; i++)

{

for (zj = -1, j = 1; j < n; j++)

{

if (!a[v[j]] && (zj < 0 || w[j] > w[zj]))

{

zj = j;

}

}

a[v[zj]] = 1;

if (i == n - 1)

{

if (best > w[zj])

{

best = w[zj];

}

for (i = 0; i < n; i++)

{

g[v[i]][pv] = g[pv][v[i]] += g[v[zj]][v[i]];

}

v[zj] = v[--n];

break;

}

pv = v[zj];

for (j = 1; j < n; j++)

{

if(!a[v[j]])

{

w[j] += g[v[zj]][v[j]];

}

}

}

}

return best;

}

### 最大流

##### Dinic算法

/\*

\* Dinic 最大流 O(V^2 \* E)

\* INIT: ne=2; head[]置为0; addedge()加入所有弧;

\* CALL: flow(n, s, t);

\*/

#define typec int // type of cost

const typec inf = 0x3f3f3f3f; // max of cost

const typec E = 10010;

const typec N = 1010;

struct edge

{

int x, y, nxt;

typec c;

} bf[E];

int ne, head[N], cur[N], ps[N], dep[N];

void addedge(int x, int y, typec c)

{ // add an arc(x->y, c); vertex:0~n-1;

bf[ne].x = x;

bf[ne].y = y;

bf[ne].c = c;

bf[ne].nxt = head[x];

head[x] = ne++;

bf[ne].x = y;

bf[ne].y = x;

bf[ne].c = 0;

bf[ne].nxt = head[y];

head[y] = ne++;

return ;

}

typec flow(int n, int s, int t)

{

typec tr, res = 0;

int i, j, k, f, r, top;

while (1)

{

memset(dep, -1, n \* sizeof(int));

for (f = dep[ps[0] = s] = 0, r = 1; f != r;)

{

for (i = ps[f++], j = head[i]; j; j = bf[j].nxt)

{

if (bf[j].c && -1 == dep[k = bf[j].y])

{

dep[k] = dep[i] + 1;

ps[r++] = k;

if (k == t)

{

f = r;

break;

}

}

}

}

if (-1 == dep[t])

{

break;

}

memcpy(cur, head, n \* sizeof(int));

for (i = s, top = 0; ;)

{

if (i == t)

{

for (k = 0, tr = inf; k < top; ++k)

{

if (bf[ps[k]].c < tr)

{

tr = bf[ps[f = k]].c;

}

}

for (k = 0; k < top; ++k)

{

bf[ps[k]].c -= tr, bf[ps[k]^1].c += tr;

}

res += tr;

i = bf[ps[top = f]].x;

}

for (j = cur[i]; cur[i]; j = cur[i] = bf[cur[i]].nxt)

{

if (bf[j].c && dep[i] + 1 == dep[bf[j].y])

{

break;

}

}

if (cur[i])

{

ps[top++] = cur[i];

i = bf[cur[i]].y;

}

else

{

if (0 == top)

{

break;

}

dep[i] = -1;

i = bf[ps[--top]].x;

}

}

}

return res;

}

##### HLPP算法

/\*

\* HLPP 最大流 O(V^3)

\* INIT: network g; g.build(nv, ne);

\* CALL: res = g.maxflow(s, t);

\* 注意: 不要加入指向源点的边, 可能死循环.

\*/

#define typef int // type of flow

const typef inf = 0x3f3f3f3f; // max of flow

const typef N = 10010;

typef minf(typef a, typef b)

{

return a < b ? a : b;

}

struct edge

{

int u, v;

typef cuv, cvu, flow;

edge (int x = 0, int y = 0, typef cu = 0, typef cv = 0, typef f = 0) : u(x), v(y), cuv(cu), cvu(cv), flow(f) {}

int other(int p)

{

return p == u ? v : u;

}

typef cap(int p)

{

return p == u ? cuv - flow : cvu + flow;

}

void addflow(int p, typef f)

{

flow += (p == u ? f : -f);

return ;

}

};

struct vlist

{

int lv, next[N], idx[2 \* N], v;

void clear(int cv)

{

v = cv;

lv = -1;

memset(idx, -1, sizeof(idx));

}

void insert(int n, int h)

{

next[n] = idx[h];

idx[h] = n;

if (lv < h)

{

lv = h;

}

}

int remove()

{

int r = idx[lv];

idx[lv] = next[idx[lv]];

while (lv >= 0 && idx[lv] == -1)

{

lv--;

}

return r;

}

bool empty()

{

return lv < 0;

}

};

struct network

{

vector<edge>eg;

vector<edge\*>net[N];

vlist list;

typef e[N];

int v, s, t, h[N], hn[2 \* N], cur[N];

void push(int);

void relabel(int);

void build(int, int);

typef maxflow(int, int);

};

void network::push(int u)

{

edge\* te = net[u][cur[u]];

typef ex = minf(te->cap(u), e[u]);

int p = te->other(u);

if (e[p] == 0 && p != t)

{

list.insert(p, h[p]);

}

te->addflow(u, ex);

e[u] -= ex;

e[p] += ex;

return ;

}

void network::relabel(int u)

{

int i, p, mh = 2 \* v, oh = h[u];

for (i = (int)net[u].size() - 1; i >= 0; i--)

{

p = net[u][i]->other(u);

if (net[u][i]->cap(u) != 0 && mh > h[p] + 1)

{

mh = h[p] + 1;

}

}

hn[h[u]]--;

hn[mh]++;

h[u] = mh;

cur[u] = (int)net[u].size() - 1;

if (hn[oh] != 0 || oh >= v + 1)

{

return ;

}

for (i = 0; i < v; i++)

{

if (h[i] > oh && h[i] <= v && i != s)

{

hn[h[i]]--;

hn[v+1]++;

h[i] = v + 1;

}

}

return ;

}

typef network::maxflow(int ss, int tt)

{

s = ss; t = tt;

int i, p, u; typef ec;

for (i = 0; i < v; i++)

{

net[i].clear();

}

for (i = (int)eg.size() - 1; i >= 0; i--)

{

net[eg[i].u].push\_back(&eg[i]);

net[eg[i].v].push\_back(&eg[i]);

}

memset(h, 0, sizeof(h));

memset(hn, 0, sizeof(hn));

memset(e, 0, sizeof(e));

e[s] = inf;

for (i = 0; i < v; i++)

{

h[i] = v;

}

queue<int> q;

q.push(t);

h[t] = 0;

while (!q.empty())

{

p = q.front();

q.pop();

for (i = (int)net[p].size() - 1; i >= 0; i--)

{

u = net[p][i]->other(p);

ec = net[p][i]->cap(u);

if (ec != 0 && h[u] == v && u != s)

{

h[u] = h[p] + 1;

q.push(u);

}

}

}

for (i = 0; i < v; i++)

{

hn[h[i]]++;

}

for (i = 0; i < v; i++)

{

cur[i] = (int)net[i].size()-1;

}

list.clear(v);

for (; cur[s] >= 0; cur[s]--)

{

push(s);

}

while (!list.empty())

{

for (u = list.remove(); e[u] > 0; )

{

if (cur[u] < 0)

{

relabel(u);

}

else if (net[u][cur[u]]->cap(u) > 0 && h[u] == h[net[u][cur[u]]->other(u)] + 1)

{

push(u);

}

else

{

cur[u]--;

}

}

}

return e[t];

}

void network::build(int n, int m)

{

v = n;

eg.clear();

int a, b, i;

typef l;

for (i = 0; i < m; i++)

{

cin >> a >> b >> l;

eg.push\_back(edge(a, b, l, 0)); // vertex: 0 ~ n-1

}

return ;

}

### 最小费用流

/\*

\* 最小费用流 O(V \* E \* f)

\* INIT: network g; g.build(v, e);

\* CALL: g.mincost(s, t); flow=g.flow; cost=g.cost;

\* 注意: SPFA增广, 实际复杂度远远小于O(V \* E);

\*/

#define typef int // type of flow

#define typec int // type of dis

const typef inff = 0x3f3f3f3f; // max of flow

const typec infc = 0x3f3f3f3f; // max of dis

const int E = 10010;

const int N = 1010;

struct network

{

int nv, ne, pnt[E], nxt[E];

int vis[N], que[N], head[N], pv[N], pe[N];

typef flow, cap[E];

typec cost, dis[E], d[N];

void addedge(int u, int v, typef c, typec w)

{

pnt[ne] = v;

cap[ne] = c;

dis[ne] = +w;

nxt[ne] = head[u];

head[u] = (ne++);

pnt[ne] = u;

cap[ne] = 0;

dis[ne] = -w;

nxt[ne] = head[v];

head[v] = (ne++);

}

int mincost(int src, int sink)

{

int i, k, f, r;

typef mxf;

for (flow = 0, cost = 0; ;)

{

memset(pv, -1, sizeof(pv));

memset(vis, 0, sizeof(vis));

for (i = 0; i < nv; ++i)

{

d[i] = infc;

}

d[src] = 0;

pv[src] = src;

vis[src] = 1;

for (f = 0, r = 1, que[0] = src; r != f;)

{

i = que[f++];

vis[i] = 0;

if (N == f)

{

f = 0;

}

for (k = head[i]; k != -1; k = nxt[k])

{

if(cap[k] && dis[k]+d[i] < d[pnt[k]])

{

d[pnt[k]] = dis[k] + d[i];

if (0 == vis[pnt[k]])

{

vis[pnt[k]] = 1;

que[r++] = pnt[k];

if (N == r)

{

r = 0;

}

}

pv[pnt[k]] = i;

pe[pnt[k]] = k;

}

}

}

if (-1 == pv[sink])

{

break;

}

for (k = sink, mxf = inff; k != src; k = pv[k])

{

if (cap[pe[k]] < mxf)

{

mxf = cap[pe[k]];

}

}

flow += mxf;

cost += d[sink] \* mxf;

for (k = sink; k != src; k = pv[k])

{

cap[pe[k]] -= mxf;

cap[pe[k] ^ 1] += mxf;

}

}

return cost;

}

void build(int v, int e)

{

nv = v;

ne = 0;

memset(head, -1, sizeof(head));

int x, y;

typef f;

typec w;

for (int i = 0; i < e; ++i)

{

cin >> x >> y >> f >> w; // vertex: 0 ~ n-1

addedge(x, y, f, w); // add arc (u->v, f, w)

}

}

} g;

/\*

\* 最小费用流 O(V^2 \* f)

\* INIT: network g; g.build(nv, ne);

\* CALL: g.mincost(s, t); flow=g.flow; cost=g.cost;

\* 注意: 网络中弧的cost需为非负. 若存在负权, 进行如下转化:

\* 首先如果原图有负环, 则不存在最小费用流. 那么可以用Johnson

\* 重标号技术把所有边变成正权，以后每次增广后进行维护，算法如下：

\* 1、用bellman-ford求s到各点的距离phi[];

\* 2、以后每求一次最短路，设s到各点的最短距离为dis[];

\* for i = 1 to v do

\* phi[v] += dis[v];

\* 下面的代码已经做了第二步，如果原图有负权，添加第一步即可。

\*/

#define typef int // type of flow

#define typec int // type of cost

const typef inff = 0x3f3f3f3f; // max of flow

const typec infc = 0x3f3f3f3f; // max of cost

const int E = 10010;

const int N = 1010;

struct edge

{

int u, v;

typef cuv, cvu, flow;

typec cost;

edge (int x, int y, typef cu, typef cv, typec cc) :u(x), v(y), cuv(cu), cvu(cv), flow(0), cost(cc){}

int other(int p)

{

return p == u ? v : u;

}

typef cap(int p)

{

return p == u ? cuv-flow : cvu+flow;

}

typec ecost(int p)

{

if (flow == 0)

{

return cost;

}

else if (flow > 0)

{

return p == u ? cost : -cost;

}

else

{

return p == u ? -cost : cost;

}

}

void addFlow(int p, typef f)

{

flow += (p == u ? f : -f);

}

};

struct network

{

vector<edge> eg;

vector<edge\*> net[N];

edge \*prev[N];

int v, s, t, pre[N], vis[N];

typef flow;

typec cost, dis[N], phi[N];

bool dijkstra();

void build(int nv, int ne);

typec mincost(int, int);

};

bool network::dijkstra()

{

// 使用O(E \* logV)的Dij可降低整体复杂度至 O(E \* logV \* f)

int i, j, p, u = 0;

typec md, cw;

for (i = 0; i < v; i++)

{

dis[i] = infc;

}

dis[s] = 0;

prev[s] = 0;

pre[s] = -1;

memset(vis, 0, v \* sizeof(int));

for (i = 1; i < v; i++)

{

for (md = infc, j = 0; j < v; j++)

{

if (!vis[j] && md > dis[j])

{

md = dis[j];

u = j;

}

}

if (md == infc)

{

break;

}

for (vis[u] = 1, j = (int)net[u].size() - 1; j >= 0; j--)

{

edge \*ce = net[u][j];

if (ce->cap(u) > 0)

{

p = ce->other(u);

cw = ce->ecost(u) + phi[u] - phi[p];

// !! assert(cw >= 0);

if (dis[p] > dis[u] + cw)

{

dis[p] = dis[u] + cw;

prev[p] = ce;

pre[p] = u;

}

}

}

}

return infc != dis[t];

}

typec network::mincost(int ss, int tt)

{

s = ss;

t = tt;

int i, c;

typef ex;

flow = cost = 0;

memset(phi, 0, sizeof(phi));

// !! 若原图含有负消费的边, 在此处运行Bellmanford

// 将phi[i](0 <= i <= n - 1)置为mindist(s, i).

for (i = 0; i < v; i++)

{

net[i].clear();

}

for (i = (int)eg.size() - 1; i >= 0; i--)

{

net[eg[i].u].push\_back(&eg[i]);

net[eg[i].v].push\_back(&eg[i]);

}

while (dijkstra())

{

for (ex = inff, c = t; c != s; c = pre[c])

{

if (ex > prev[c]->cap(pre[c]))

{

ex = prev[c]->cap(pre[c]);

}

}

for (c = t; c != s; c = pre[c])

{

prev[c]->addFlow(pre[c], ex);

}

flow += ex;

cost += ex \* (dis[t] + phi[t]);

for (i = 0; i < v; i++)

{

phi[i] += dis[i];}

}

return cost;

}

void network::build(int nv, int ne)

{

eg.clear();

v = nv;

int x, y;

typef f;

typec c;

for (int i = 0; i < ne; ++i)

{

cin >> x >> y >> f >> c;

eg.push\_back(edge(x, y, f, 0, c));

}

return ;

}

### 有上下界的流

##### 有上下界的最小（最大）流

/\*

\* 有上下界的最小(最大)流

\* INIT: up[][]为容量上界; low[][]为容量下界;

\* CALL: mf = limitflow(n,src,sink); flow[][]为流量分配;

\* 另附: 循环流问题

\* 描述: 无源无汇的网络N,设N是具有基础有向图D=(V,A)的网络.

\* l和c分别为容量下界和容量上界. 如果定义在A上的函数

\* f满足: f(v, V) = f(V, v). V中任意顶点v,

\* l(a)<=f(a)<=c(a),则称f为网络N的循环流.

\* 解法: 添加一个源s和汇t,对于每个下限容量l不为0的边(u, v),

\* 将其下限去掉,上限改为c-l,增加两条边(u, t),(s, v),

\* 容量均为l.原网络存在循环流等价于新网络最大流是满流.

\*/

const int inf = 0x3f3f3f3f;

const int N = 1010;

int up[N][N], low[N][N], flow[N][N];

int pv[N], que[N], d[N];

void maxflow(int n, int src, int sink)

{

// BFS增广, O(E \* maxflow)

int p, q, t, i, j;

do

{

for (i = 0; i < n; pv[i++] = 0);

pv[t = src] = src + 1;

d[t] = inf;

for (p = q = 0; p <= q && !pv[sink]; t = que[p++])

{

for (i = 0; i < n; i++)

{

if (!pv[i] && up[t][i] && (j = up[t][i] - flow[t][i]) > 0)

{

pv[que[q++] = i] = +t + 1, d[i] = d[t] < j ? d[t] : j;

}

else if (!pv[i] && up[i][t] && (j = flow[i][t]) > 0)

{

pv[que[q++] = i] = -t - 1, d[i] = d[t] < j ? d[t] : j;

}

}

}

for (i = sink; pv[i] && i != src;)

{

if (pv[i] > 0)

{

flow[pv[i] - 1][i] += d[sink], i = pv[i] - 1;

}

else

{

flow[i][-pv[i] - 1] -= d[sink], i = -pv[i] - 1;

}

}

}

while (pv[sink]);

return ;

}

int limitflow(int n, int src, int sink)

{

int i, j, sk, ks;

if (src == sink)

{

return inf;

}

up[n][n + 1] = up[n + 1][n] = up[n][n] = up[n + 1][n + 1] = 0;

for (i = 0; i < n; i++)

{

up[n][i] = up[i][n] = up[n+1][i] = up[i][n+1] = 0;

for (j = 0; j < n; j++)

{

up[i][j] -= low[i][j];

up[n][i] += low[j][i];

up[i][n + 1] += low[i][j];

}

}

sk = up[src][sink];

ks = up[sink][src];

up[src][sink] = up[sink][src] = inf;

maxflow(n + 2, n, n + 1);

for (i = 0; i < n; i++)

{

if (flow[n][i] < up[n][i])

{

return -1;

}

}

flow[src][sink] = flow[sink][src] = 0;

up[src][sink] = sk;

up[sink][src] = ks; // !min:src<-sink; max:src->sink;

maxflow(n, sink, src);

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

up[i][j] += low[i][j];

flow[i][j] += low[i][j];

}

}

for (j = i = 0; i < n; j += flow[src][i++]);

return j;

}

### 最佳边割集

#define MAXN 100

#define inf 1000000000

int max\_flow(int n, int mat[][MAXN], int source, int sink)

{

int v[MAXN], c[MAXN], p[MAXN], ret = 0, i, j;

for (;;)

{

for (i = 0; i < n; i++)

{

v[i] = c[i] = 0;

}

for (c[source] = inf; ;)

{

for (j = -1, i = 0; i < n; i++)

{

if (!v[i] && c[i] && (j == -1 || c[i] > c[j]))

{

j = i;

}

}

if (j < 0)

{

return ret;

}

if (j == sink)

{

break;

}

for (v[j] = 1, i = 0; i < n; i++)

{

if (mat[j][i] > c[i] && c[j] > c[i])

{

c[i] = mat[j][i] < c[j] ? mat[j][i] : c[j], p[i] = j;

}

}

}

for (ret += j = c[i = sink]; i != source; i = p[i])

{

mat[p[i]][i] -= j, mat[i][p[i]] += j;

}

}

}

int best\_edge\_cut(int n, int mat[][MAXN], int source, int sink, int set[][2], int &mincost)

{

int m0[MAXN][MAXN], m[MAXN][MAXN], i, j, k, l, ret = 0, last;

if (source == sink)

{

return -1;

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

m0[i][j] = mat[i][j];

}

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

m[i][j] = m0[i][j];

}

}

mincost = last = max\_flow(n, m, source, sink);

for (k = 0; k < n && last; k++)

{

for (l = 0; l < n && last; l++)

{

if (m0[k][l])

{

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

m[k][l] = 0;

if (max\_flow(n, m, source, sink) == last - mat[k][l])

{

set[ret][0] = k;

set[ret++][1] = l;

m0[k][l] = 0;

last -= mat[k][l];

}

}

}

}

return ret;

}

### 最佳点割集

#define MAXN 100

#define inf 1000000000

int max\_flow(int n, int mat[][MAXN], int source, int sink)

{

int v[MAXN], c[MAXN], p[MAXN], ret = 0, i, j;

for (;;)

{

for (i = 0; i < n; i++)

{

v[i] = c[i] = 0;

}

for (c[source] = inf; ;)

{

for (j = -1, i = 0; i < n; i++)

{

if (!v[i] && c[i] && (j == -1 || c[i] > c[j]))

{

j = i;

}

}

if (j < 0)

{

return ret;

}

if (j == sink)

{

break;

}

for (v[j] = 1, i = 0; i < n; i++)

{

if (mat[j][i] > c[i] && c[j] > c[i])

{

c[i] = mat[j][i] < c[j] ? mat[j][i] : c[j], p[i] = j;

}

}

}

for (ret += j = c[i = sink]; i != source; i = p[i])

{

mat[p[i]][i] -= j, mat[i][p[i]] += j;

}

}

}

int best\_vertex\_cut(int n, int mat[][MAXN], int \*cost, int source, int sink, int \*set, int &mincost)

{

int m0[MAXN][MAXN], m[MAXN][MAXN], i, j, k, ret = 0, last;

if (source == sink || mat[source][sink])

{

return -1;

}

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m0[i][j] = 0;

}

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (mat[i][j])

{

m0[i][n + j] = inf;

}

}

}

for (i = 0; i < n; i++)

{

m0[n + i][i] = cost[i];

}

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

mincost = last = max\_flow(n + n, m, source, n + sink);

for (k = 0; k < n && last; k++)

{

if (k != source && k != sink)

{

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

m[n + k][k] = 0;

if (max\_flow(n + n, m, source, n + sink) == last - cost[k])

{

set[ret++] = k;

m0[n + k][k] = 0;

last -= cost[k];

}

}

}

return ret;

}

### 最小边割集

#define MAXN 100

#define inf 1000000000

int max\_flow(int n, int mat[][MAXN], int source, int sink)

{

int v[MAXN], c[MAXN], p[MAXN], ret = 0, i, j;

for (;;)

{

for (i = 0; i < n; i++)

{

v[i] = c[i] = 0;

}

for (c[source] = inf; ;)

{

for (j = -1, i = 0; i < n; i++)

{

if (!v[i] && c[i] && (j == -1 || c[i] > c[j]))

{

j = i;

}

}

if (j < 0)

{

return ret;

}

if (j == sink)

{

break;

}

for (v[j] = 1, i = 0; i < n; i++)

{

if (mat[j][i] > c[i] && c[j] > c[i])

{

c[i] = mat[j][i] < c[j] ? mat[j][i] : c[j], p[i] = j;

}

}

}

for (ret += j = c[i = sink]; i != source; i = p[i])

{

mat[p[i]][i] -= j, mat[i][p[i]] += j;

}

}

}

int min\_edge\_cut(int n, int mat[][MAXN], int source, int sink, int set[][2])

{

int m0[MAXN][MAXN], m[MAXN][MAXN], i, j, k, l, ret = 0, last;

if (source == sink)

{

return -1;

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

m0[i][j] = (mat[i][j] != 0);

}

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

m[i][j] = m0[i][j];

}

}

last = max\_flow(n, m, source, sink);

for (k = 0; k < n && last; k++)

{

for (l = 0; l < n && last; l++)

{

if (m0[k][l])

{

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

m[k][l] = 0;

if (max\_flow(n, m, source, sink) < last)

{

set[ret][0] = k;

set[ret++][1] = l;

m0[k][l] = 0;

last--;

}

}

}

}

return ret;

}

### 最小点割集

/\*

\* 最小点割集(点连通度)

\*/

#define MAXN 100

#define inf 1000000000

int max\_flow(int n, int mat[][MAXN], int source, int sink)

{

int v[MAXN], c[MAXN], p[MAXN], ret = 0, i, j;

for (; ;)

{

for (i = 0; i < n; i++)

{

v[i] = c[i] = 0;

}

for (c[source] = inf; ;)

{

for (j = -1, i = 0; i < n; i++)

{

if (!v[i] && c[i] && (j == -1 || c[i] > c[j]))

{

j = i;

}

}

if (j < 0)

{

return ret;

}

if (j == sink)

{

break;

}

for (v[j] = 1, i = 0; i < n; i++)

{

if (mat[j][i] > c[i] && c[j] > c[i])

{

c[i] = mat[j][i] < c[j] ? mat[j][i] : c[j], p[i] = j;

}

}

}

for (ret += j = c[i = sink]; i != source; i = p[i])

{

mat[p[i]][i] -= j, mat[i][p[i]] += j;

}

}

}

int min\_vertex\_cut(int n, int mat[][MAXN], int source, int sink, int \*set)

{

int m0[MAXN][MAXN], m[MAXN][MAXN], i, j, k, ret = 0, last;

if (source == sink || mat[source][sink])

{

return -1;

}

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m0[i][j] = 0;

}

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (mat[i][j])

{

m0[i][n + j] = inf;

}

}

}

for (i = 0; i < n; i++)

{

m0[n + i][i]=1;

}

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

last = max\_flow(n + n, m, source, n + sink);

for (k = 0; k < n && last; k++)

{

if (k != source && k != sink)

{

for (i = 0; i < n + n; i++)

{

for (j = 0; j < n + n; j++)

{

m[i][j] = m0[i][j];

}

}

m[n+k][k] = 0;

if (max\_flow(n + n, m, source, n + sink) < last)

{

set[ret++] = k;

m0[n+k][k] = 0;

last--;

}

}

}

return ret;

}

### 最小覆盖问题

##### 最小路径覆盖

最小路径覆盖O(n^3)路径覆盖:就是在图中找一些路经,使之覆盖了图中的所有顶点,且任何一个顶点有且只有一条路径与之关联。

最小路径覆盖:就是找出最少的路径条数,使之成为P的一个路径覆盖。

路径覆盖与二分图匹配的关系:最小路径覆盖=|P|-最大匹配数;其中最大匹配数的求法是把P中的每个顶点pi分成两个顶点pi’与pi”,如果在p中存在一条pi到pj的边,那么在二分图P’中就有一条连接pi’与pj”的有向边(求二分图匹配时必须是单向边);这里pi’就是p中pi的出边,pj”就是p中pj的一条入边;

有向图: 最小路径覆盖=|P|-最大匹配数;

无向图: 最小路径覆盖=|P|-最大匹配数/2;

##### 最小点集覆盖

结论:一个二分图中的最大匹配数等于这个图中的最小点覆盖数。

# Structure 数据结构

### 划分树

/\*

\* 划分树(查询区间第k大)

\*/

const int MAXN = 100010;

int tree[20][MAXN]; // 表示每层每个位置的值

int sorted[MAXN]; // 已经排序好的数

int toleft[20][MAXN]; // toleft[p][i]表示第i层从1到i有数分入左边

void build(int l, int r, int dep)

{

if (l == r)

{

return;

}

int mid = (l + r) >> 1;

int same = mid - l + 1; // 表示等于中间值而且被分入左边的个数

for (int i = l; i <= r; i++) // 注意是l,不是one

{

if (tree[dep][i] < sorted[mid])

{

same--;

}

}

int lpos = l;

int rpos = mid + 1;

for (int i = l; i <= r; i++)

{

if (tree[dep][i] < sorted[mid])

{

tree[dep + 1][lpos++] = tree[dep][i];

}

else if (tree[dep][i] == sorted[mid] && same > 0)

{

tree[dep + 1][lpos++] = tree[dep][i];

same--;

}

else

{

tree[dep + 1][rpos++] = tree[dep][i];

}

toleft[dep][i] = toleft[dep][l - 1] + lpos - l;

}

build(l, mid, dep + 1);

build(mid + 1, r, dep + 1);

return ;

}

// 查询区间第k大的数,[L,R]是大区间,[l,r]是要查询的小区间

int query(int L, int R, int l, int r, int dep, int k)

{

if(l == r)

{

return tree[dep][l];

}

int mid = (L + R) >> 1;

int cnt = toleft[dep][r] - toleft[dep][l - 1];

if (cnt >= k)

{

int newl = L + toleft[dep][l - 1] - toleft[dep][L - 1];

int newr = newl + cnt - 1;

return query(L, mid, newl, newr, dep + 1, k);

}

else

{

int newr = r + toleft[dep][R] - toleft[dep][r];

int newl = newr - (r - l - cnt);

return query(mid + 1, R, newl, newr, dep + 1, k - cnt);

}

}

int main()

{

int n, m;

while (scanf("%d%d", &n, &m) == 2)

{

memset(tree, 0, sizeof(tree));

for (int i = 1; i <= n; i++)

{

scanf("%d", &tree[0][i]);

sorted[i] = tree[0][i];

}

sort(sorted + 1, sorted + n + 1);

build(1, n, 0);

int s, t, k;

while(m--)

{

scanf("%d%d%d", &s, &t, &k);

printf("%d\n", query(1, n, s, t, 0, k));

}

}

return 0;

}

### 左偏树

/\*

\* 合并复杂度 O(log N)

\* INIT: init()读入数据并进行初始化;

\* CALL: merge() 合并两棵左偏树;

\* ins() 插入一个新节点;

\* top() 取得最小结点;

\* pop() 取得并删除最小结点;

\* del() 删除某结点;

\* add() 增/减一个结点的键值;

\* iroot() 获取结点i的根;

\*/

#define typec int // type of key val

const int na = -1;

const int N = 1010;

struct node

{

typec key;

int l, r, f, dist;

} tr[N];

int iroot(int i)

{ // find i's root

if (i == na)

{

return i;

}

while (tr[i].f != na)

{

i = tr[i].f;

}

return i;

}

int merge(int rx, int ry)

{

// two root: rx, ry

if (rx == na)

{

return ry;

}

if (ry == na)

{

return rx;

}

if (tr[rx].key > tr[ry].key)

{

swap(rx, ry);

}

int r = merge(tr[rx].r, ry);

tr[rx].r = r;

tr[r].f = rx;

if (tr[r].dist > tr[tr[rx].l].dist)

{

swap(tr[rx].l, tr[rx].r);

}

if (tr[rx].r == na)

{

tr[rx].dist = 0;

}

else

{

tr[rx].dist = tr[tr[rx].r].dist + 1;

}

return rx; // return new root

}

int ins(int i, typec key, int root)

{ // add a new node(i, key)

tr[i].key = key;

tr[i].l = tr[i].r = tr[i].f = na;

tr[i].dist = 0;

return root = merge(root, i); // return new root

}

int del(int i)

{ // delete node i

if (i == na)

{

return i;

}

int x, y, l, r;

l = tr[i].l;

r = tr[i].r;

y = tr[i].f;

tr[i].l = tr[i].r = tr[i].f = na;

tr[x = merge(l, r)].f = y;

if (y != na && tr[y].l == i)

{

tr[y].l = x;

}

if (y != na && tr[y].r == i)

{

tr[y].r = x;

}

for (; y != na; x = y, y = tr[y].f)

{

if (tr[tr[y].l].dist < tr[tr[y].r].dist)

{

swap(tr[y].l, tr[y].r);

}

if (tr[tr[y].r].dist + 1 == tr[y].dist)

{

break;

}

tr[y].dist = tr[tr[y].r].dist + 1;

}

if (x != na) // return new root

{

return iroot(x);

}

else return iroot(y);

}

node top(int root)

{

return tr[root];

}

node pop(int &root)

{

node out = tr[root];

int l = tr[root].l, r = tr[root].r;

tr[root].l = tr[root].r = tr[root].f = na;

tr[l].f = tr[r].f = na;

root = merge(l, r);

return out;

}

int add(int i, typec val) // tr[i].key += val

{

if (i == na)

{

return i;

}

if (tr[i].l == na && tr[i].r == na && tr[i].f == na)

{

tr[i].key += val;

return i;

}

typec key = tr[i].key + val;

int rt = del(i);

return ins(i, key, rt);

}

void init(int n)

{

for (int i = 1; i <= n; i++)

{

scanf("%d", &tr[i].key); // %d: type of key

tr[i].l = tr[i].r = tr[i].f = na;

tr[i].dist = 0;

}

return ;

}

### 线段树

##### 求矩形并的面积(线段树+离散化+扫描线)

Each test case starts with a line containing a single integer n (1 <= n <= 100) of available maps.

The n following lines describe one map each.

Each of these lines contains four numbers x1、y1、x2、y2 (0 <= x1 < x2 <= 100000; 0 <= y1 < y2 <= 100000), not necessarily integers.

The values (x1， y1) and (x2，y2) are the coordinates of the

top-left resp.

Bottom-Right corner of the mapped area.

/\*

\* 本题中的坐标是浮点类型的, 故不能将坐标直接离散.我们必须为它们建立一个对应关系,

\* 用一个整数去对应一个浮点数这样的对应关系在本题的数组y[]中。

\*/

struct node

{

int st, ed, c; // c: 区间被覆盖的层数, m: 区间的测度

double m;

} ST[802];

struct line

{

double x, y1, y2; // 纵方向直线, x:直线横坐标, y1 y2:直线上的下面与上面的两个纵坐标

bool s; // s = 1 : 直线为矩形的左边, s = 0:直线为矩形的右边

} Line[205];

double y[205], ty[205]; // y[]整数与浮点数的对应数组; ty[]:用来求y[]的辅助数组

void build(int root, int st, int ed)

{

ST[root].st = st;

ST[root].ed = ed;

ST[root].c = 0;

ST[root].m = 0;

if (ed - st > 1)

{

int mid = (st + ed) / 2;

build(root \* 2, st, mid);

build(root \* 2 + 1, mid, ed);

}

return ;

}

void updata(int root)

{

if (ST[root].c > 0)

{ // 将线段树上区间的端点分别映射到y[]数组所对应的浮点数上,由此计算出测度

ST[root].m = y[ST[root].ed - 1] - y[ST[root].st - 1];

}

else if (ST[root].ed - ST[root].st == 1)

{

ST[root].m = 0;

}

else

{

ST[root].m = ST[root \* 2].m + ST[root \* 2 + 1].m;

}

return ;

}

void insert(int root, int st, int ed)

{

if (st <= ST[root].st && ST[root].ed <= ed)

{

ST[root].c++;

updata(root);

return ;

}

if (ST[root].ed - ST[root].st == 1)

{

return ; // 不出错的话 这句话就是冗余的

}

int mid = (ST[root].ed + ST[root].st) / 2;

if (st < mid)

{

insert(root \* 2, st, ed);

}

if (ed > mid)

{

insert(root \* 2 + 1, st, ed);

}

updata(root);

return ;

}

void Delete(int root, int st, int ed)

{

if (st <= ST[root].st && ST[root].ed <= ed)

{

ST[root].c--;

updata(root);

return ;

}

if (ST[root].ed - ST[root].st == 1)

{

return ; // 不出错的话 这句话就是冗余的

}

int mid = (ST[root].st + ST[root].ed) / 2;

if (st < mid)

{

Delete(root \* 2, st, ed);

}

if (ed > mid)

{

Delete(root \* 2 + 1, st, ed);

}

updata(root);

return ;

}

int Correspond(int n, double t)

{

// 二分查找出浮点数t在数组y[]中的位置(此即所谓的映射关系)

int low, high, mid;

low = 0;

high = n - 1;

while (low < high)

{

mid = (low + high) / 2;

if (t > y[mid])

{

low = mid + 1;

}

else

{

high = mid;

}

}

return high + 1;

}

bool cmp(line l1, line l2)

{

return l1.x < l2.x;

}

int main()

{

int n, i, num, l, r, c = 0;

double area, x1, x2, y1, y2;

while (cin >> n, n)

{

for (i = 0; i < n; i++)

{

cin >> x1 >> y1 >> x2 >> y2;

Line[2 \* i].x = x1;

Line[2 \* i].y1 = y1;

Line[2 \* i].y2 = y2;

Line[2 \* i].s = 1;

Line[2 \* i + 1].x = x2;

Line[2 \* i + 1].y1 = y1;

Line[2 \* i + 1].y2 = y2;

Line[2 \* i + 1].s = 0;

ty[2 \* i] = y1;

ty[2 \* i + 1] = y2;

}

n <<= 1;

sort(Line, Line + n, cmp);

sort(ty, ty + n);

y[0] = ty[0];

// 处理数组ty[]使之不含重覆元素,得到新的数组存放到数组y[]中

for (i = num = 1; i < n; i++)

{

if (ty[i] != ty[i - 1])

{

y[num++] = ty[i];

}

}

build(1, 1, num); // 树的叶子节点与数组y[]中的元素个数相同,以便建立一一对应的关系

area = 0;

for (i = 0; i < n - 1; i++)

{ // 由对应关系计算出线段两端在树中的位置

l = Correspond(num, Line[i].y1);

r = Correspond(num, Line[i].y2);

if (Line[i].s) // 插入矩形的左边

{

insert(1, l, r);

}

else // 删除矩形的右边

{

Delete(1, l, r);

}

area += ST[1].m \* (Line[i + 1].x - Line[i].x);

}

cout << "Test case #" << ++c << endl << "Total explored area: ";

cout << fixed << setprecision(2) << area << endl << endl; // 需要引入iomanip头文件

}

return 0;

}

##### 求矩形并的周长（线段树+离散化+扫描线）

The first line contains the number of rectangles pasted on the wall.

In each of the subsequent lines, one can find the integer coordinates of the lower left vertex and the upper right vertex of each rectangle.

The values of those coordinates are given as ordered pairs consisting of an x-coordinate followed by a y-coordinate. 0 <= number of rectangles < 5000.

All coordinates are in the range [-10000,10000] and any existing rectangle has a positive area.

struct node

{

int st, ed, m, lbd, rbd;

int sequence\_line, count;

} ST[40005];

void build(int st, int ed, int v) // 建树,区间为[st, ed]

{

ST[v].st = st;

ST[v].ed = ed;

ST[v].m = ST[v].lbd = ST[v].rbd = 0;

ST[v].sequence\_line = ST[v].count = 0;

if (ed - st > 1)

{

int mid = (st + ed) / 2;

build(st, mid, 2 \* v + 1);

build(mid, ed, 2 \* v + 2);

}

return ;

}

void UpData(int v) // 更新结点区间的测度

{

if (ST[v].count > 0)

{

ST[v].m = ST[v].ed - ST[v].st;

ST[v].lbd = ST[v].rbd = 1;

ST[v].sequence\_line = 1;

return ;

}

if (ST[v].ed - ST[v].st == 1)

{

ST[v].m = 0;

ST[v].lbd = ST[v].rbd = 0;

ST[v].sequence\_line = 0;

}

else

{

int left = 2 \* v + 1, right = 2 \* v + 2;

ST[v].m = ST[left].m + ST[right].m;

ST[v].sequence\_line = ST[left].sequence\_line + ST[right].sequence\_line - (ST[left].rbd & ST[right].lbd);

ST[v].lbd = ST[left].lbd;

ST[v].rbd = ST[right].rbd;

}

return ;

}

void insert(int st, int ed, int v)

{

if (st <= ST[v].st && ed >= ST[v].ed)

{

ST[v].count++;

UpData(v);

return ;

}

int mid = (ST[v].st + ST[v].ed) / 2;

if (st < mid)

{

insert(st, ed, 2 \* v + 1);

}

if (ed > mid)

{

insert(st, ed, 2 \* v + 2);

}

UpData(v);

return ;

}

void Delete(int st, int ed, int v)

{

if (st <= ST[v].st && ed >= ST[v].ed)

{

ST[v].count--;

UpData(v);

return ;

}

int mid = (ST[v].st + ST[v].ed) / 2;

if (st < mid)

{

Delete(st, ed, 2 \* v + 1);

}

if (ed > mid)

{

Delete(st, ed, 2 \* v + 2);

}

UpData(v);

return ;

}

struct line

{

int x, y1, y2; // y1 < y2

bool d; // d=true表示该线段为矩形左边,d=false表示该线段为矩形的右边

} a[10003];

bool cmp(line t1, line t2) // 为线段排序的函数,方便从左向右的扫描

{

return t1.x < t2.x;

}

void cal\_C(int n);

int main()

{

int n, x1, x2, y1, y2, i, j, suby, upy;

while (scanf("%d",&n) != EOF)

{

j = 0;

suby = 10000;

upy = -10000;

for (i = 0; i < n; i++)

{

scanf("%d%d%d%d", &x1, &y1, &x2, &y2);

a[j].x = x1;

a[j].y1 = y1;

a[j].y2 = y2;

a[j].d = 1;

j++;

a[j].x = x2;

a[j].y1 = y1;

a[j].y2 = y2;

a[j].d = 0;

j++;

if (suby > y1)

{

suby = y1;

}

if (upy < y2)

{

upy = y2;

}

}

sort(a, a + j, cmp);

build(suby, upy, 0);

cal\_C(j);

}

return 0;

}

void cal\_C(int n)

{

int i, t2, sum = 0;

t2 = 0;

a[n] = a[n - 1];

for (i = 0; i < n; i++)

{

if (a[i].d == 1)

{

insert(a[i].y1, a[i].y2, 0);

}

else

{

Delete(a[i].y1, a[i].y2, 0);

}

sum += ST[0].sequence\_line \* (a[i + 1].x - a[i].x) \* 2;

sum += abs(ST[0].m - t2);

t2 = ST[0].m;

}

printf("%d\n", sum);

}

### 主席树

##### 查找区间有多少个不同的数

/\*

\* 给出一个序列,查询区间内有多少个不相同的数

\*/

const int MAXN = 30010;

const int M = MAXN \* 100;

int n, q, tot;

int a[MAXN];

int T[MAXN], lson[M], rson[M], c[M];

int build(int l, int r)

{

int root = tot++;

c[root] = 0;

if (l != r)

{

int mid = (l + r) >> 1;

lson[root] = build(l, mid);

rson[root] = build(mid + 1, r);

}

return root;

}

int update(int root, int pos, int val)

{

int newroot = tot++, tmp = newroot;

c[newroot] = c[root] + val;

int l = 1, r = n;

while (l < r)

{

int mid = (l + r) >> 1;

if (pos <= mid)

{

lson[newroot] = tot++;

rson[newroot] = rson[root];

newroot = lson[newroot];

root = lson[root];

r = mid;

}

else

{

rson[newroot] = tot++;

lson[newroot] = lson[root];

newroot = rson[newroot];

root = rson[root];

l = mid + 1;

}

c[newroot] = c[root] + val;

}

return tmp;

}

int query(int root, int pos)

{

int ret = 0;

int l = 1, r = n;

while (pos < r)

{

int mid = (l + r) >> 1;

if (pos <= mid)

{

r = mid;

root = lson[root];

}

else

{

ret += c[lson[root]];

root = rson[root];

l = mid + 1;

}

}

return ret + c[root];

}

int main()

{

// freopen("in.txt", "r", stdin);

// freopen("out.txt", "w", stdout);

while (scanf("%d", &n) == 1)

{

tot = 0;

for (int i = 1; i <= n; i++)

{

scanf("%d", &a[i]);

}

T[n + 1] = build(1, n);

map<int,int> mp;

for (int i = n; i >= 1; i--)

{

if (mp.find(a[i]) == mp.end())

{

T[i] = update(T[i + 1], i, 1);

}

else

{

int tmp = update(T[i + 1], mp[a[i]], -1);

T[i] = update(tmp, i, 1);

}

mp[a[i]] = i;

}

scanf("%d", &q);

while (q--)

{

int l, r;

scanf("%d%d", &l, &r);

printf("%d\n", query(T[l], r));

}

}

return 0;

}

##### 静态区间第k小

const int MAXN = 100010;

const int M = MAXN \* 30;

int n, q, m, tot;

int a[MAXN], t[MAXN];

int T[MAXN], lson[M], rson[M], c[M];

void Init\_hash()

{

for (int i = 1; i <= n; i++)

{

t[i] = a[i];

}

sort(t + 1, t + 1 + n);

m = (int)(unique(t + 1, t + 1 + n) - t - 1);

}

int build(int l, int r)

{

int root = tot++; c[root] = 0;

if (l != r)

{

int mid = (l + r) >> 1;

lson[root] = build(l, mid);

rson[root] = build(mid + 1, r);

}

return root;

}

int hash\_(int x)

{

return (int)(lower\_bound(t + 1, t + 1 + m, x) - t);

}

int update(int root, int pos, int val)

{

int newroot = tot++, tmp = newroot;

c[newroot] = c[root] + val;

int l = 1, r = m;

while (l < r)

{

int mid = (l + r) >> 1;

if (pos <= mid)

{

lson[newroot] = tot++;

rson[newroot] = rson[root];

newroot = lson[newroot];

root = lson[root];

r = mid;

}

else

{

rson[newroot] = tot++;

lson[newroot] = lson[root];

newroot = rson[newroot];

root = rson[root];

l = mid + 1;

}

c[newroot] = c[root] + val;

}

return tmp;

}

int query(int left\_root, int right\_root, int k)

{

int l = 1, r = m;

while ( l < r)

{

int mid = (l + r) >> 1;

if (c[lson[left\_root]] - c[lson[right\_root]] >= k )

{

r = mid;

left\_root = lson[left\_root];

right\_root = lson[right\_root];

}

else

{

l = mid + 1;

k -= c[lson[left\_root]] - c[lson[right\_root]];

left\_root = rson[left\_root];

right\_root = rson[right\_root];

}

}

return l;

}

int main()

{

// freopen("in.txt","r",stdin);

// freopen("out.txt","w",stdout);

while (scanf("%d%d", &n, &q) == 2)

{

tot = 0;

for (int i = 1; i <= n; i++)

{

scanf("%d", &a[i]);

}

Init\_hash();

T[n + 1] = build(1, m);

for (int i = n; i; i--)

{

int pos = hash\_(a[i]);

T[i] = update(T[i + 1], pos, 1);

}

while (q--)

{

int l, r, k;

scanf("%d%d%d", &l, &r, &k);

printf("%d\n", t[query(T[l], T[r + 1], k)]);

}

}

return 0;

}

##### 树上路径点权第k大

/\*

\* LCA + 主席树

\*/

// 主席树部分

const int MAXN = 200010;

const int M = MAXN \* 40;

int n, q, m, TOT;

int a[MAXN], t[MAXN];

int T[MAXN], lson[M], rson[M], c[M];

void Init\_hash()

{

for (int i = 1; i <= n; i++)

{

t[i] = a[i];

}

sort(t + 1, t + 1 + n);

m = (int)(unique(t + 1, t + n + 1) - t - 1);

return ;

}

int build(int l, int r)

{

int root = TOT++;

c[root] = 0;

if (l != r)

{

int mid = (l + r) >> 1;

lson[root] = build(l, mid);

rson[root] = build(mid + 1, r);

}

return root;

}

int hash\_(int x)

{

return (int)(lower\_bound(t + 1, t + 1 + m, x) - t);

}

int update(int root, int pos, int val)

{

int newroot = TOT++, tmp = newroot;

c[newroot] = c[root] + val;

int l = 1, r = m;

while (l < r)

{

int mid = (l + r) >> 1;

if (pos <= mid)

{

lson[newroot] = TOT++;

rson[newroot] = rson[root];

newroot = lson[newroot];

root = lson[root];

r = mid;

}

else

{

rson[newroot] = TOT++;

lson[newroot] = lson[root];

newroot = rson[newroot];

root = rson[root];

l = mid + 1;

}

c[newroot] = c[root] + val;

}

return tmp;

}

int query(int left\_root, int right\_root, int LCA, int k)

{

int lca\_root = T[LCA];

int pos = hash\_(a[LCA]);

int l = 1, r = m;

while (l < r)

{

int mid = (l + r) >> 1;

int tmp = c[lson[left\_root]] + c[lson[right\_root]] - 2 \* c[lson[lca\_root]] + (pos >= l && pos <= mid);

if (tmp >= k)

{

left\_root = lson[left\_root];

right\_root = lson[right\_root];

lca\_root = lson[lca\_root];

r = mid;

}

else

{

k -= tmp;

left\_root = rson[left\_root];

right\_root = rson[right\_root];

lca\_root = rson[lca\_root];

l = mid + 1;

}

}

return l;

}

// LCA部分

int rmq[2 \* MAXN]; // rmq数组,就是欧拉序列对应的深度序列

struct ST

{

int mm[2 \* MAXN];

int dp[2 \* MAXN][20]; // 最小值对应的下标

void init(int n)

{

mm[0] = -1;

for (int i = 1; i <= n; i++)

{

mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

dp[i][0] = i;

}

for (int j = 1; j <= mm[n]; j++)

{

for (int i = 1; i + (1 << j) - 1 <= n; i++)

{

dp[i][j] = rmq[dp[i][j - 1]] < rmq[dp[i + (1 << (j - 1))][j - 1]] ? dp[i][j - 1] : dp[i + (1 << (j - 1))][j - 1];

}

}

return ;

}

int query(int a, int b) // 查询[a,b]之间最小值的下标

{

if (a > b)

{

swap(a, b);

}

int k = mm[b - a + 1];

return rmq[dp[a][k]] <= rmq[dp[b - (1 << k) + 1][k]] ? dp[a][k] : dp[b - (1 << k) + 1][k];

}

};

// 边的结构体定义

struct Edge

{

int to, next;

};

Edge edge[MAXN \* 2];

int tot, head[MAXN];

int F[MAXN \* 2]; // 欧拉序列,就是dfs遍历的顺序,长度为2\*n-1,下标从1开始

int P[MAXN]; // P[i]表示点i在F中第一次出现的位置

int cnt;

ST st;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

return ;

}

void addedge(int u, int v) // 加边,无向边需要加两次

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

void dfs(int u, int pre, int dep)

{

F[++cnt] = u;

rmq[cnt] = dep;

P[u] = cnt;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v == pre)

{

continue;

}

dfs(v, u, dep + 1);

F[++cnt] = u;

rmq[cnt] = dep;

}

return ;

}

void LCA\_init(int root, int node\_num) // 查询LCA前的初始化

{

cnt = 0;

dfs(root, root, 0);

st.init(2 \* node\_num - 1);

return ;

}

int query\_lca(int u, int v) // 查询u,v的lca编号

{

return F[st.query(P[u], P[v])];

}

void dfs\_build(int u, int pre)

{

int pos = hash\_(a[u]);

T[u] = update(T[pre], pos, 1);

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v == pre)

{

continue;

}

dfs\_build(v, u);

}

return ;

}

int main()

{

// freopen("in.txt", "r", stdin);

// freopen("out.txt", "w", stdout);

while (scanf("%d%d", &n, &q) == 2)

{

for (int i = 1; i <= n; i++)

{

scanf("%d", &a[i]);

}

Init\_hash();

init();

TOT = 0;

int u, v;

for (int i = 1; i < n; i++)

{

scanf("%d%d", &u, &v);

addedge(u, v);

addedge(v, u);

}

LCA\_init(1, n);

T[n + 1] = build(1, m);

dfs\_build(1, n + 1);

int k;

while (q--)

{

scanf("%d%d%d", &u, &v, &k);

printf("%d\n", t[query(T[u], T[v], query\_lca(u, v), k)]);

}

}

return 0;

}

### Trie树

##### K叉

/\*

\* INIT: init();

\* 注: tree[i][tk]>0时表示单词存在, 当然也可赋予它更多含义;

\*/

const int tk = 26, tb = 'a'; // tk叉; 起始字母为tb;

const int N = 1010; // N: 最大结点个数

int top, tree[N][tk + 1];

void init()

{

top = 1;

memset(tree[0], 0, sizeof(tree[0]));

return ;

}

int sear(char \*s) // 失败返回0

{

for (int rt = 0; rt == tree[rt][\*s - tb];)

{

if (\*(++s) == 0)

{

return tree[rt][tk];

}

}

return 0;

}

void insert(char \*s, int rank = 1)

{

int rt, nxt;

for (rt = 0; \*s; rt = nxt, ++s)

{

nxt = tree[rt][\*s - tb];

if (0 == nxt)

{

tree[rt][\*s - tb] = nxt = top;

memset(tree[top], 0, sizeof(tree[top]));

top++;

}

}

tree[rt][tk] = rank; // 1表示存在0表示不存在,也可以赋予其其他含义

}

void delt(char \*s) // 只做标记, 假定s一定存在

{

int rt = 0;

for (; \*s; ++s)

{

rt = tree[rt][\*s - tb];

}

tree[rt][tk] = 0;

return ;

}

int prefix(char \*s) // 最长前缀

{

int rt = 0, lv;

for (lv = 0; \*s; ++s, ++lv)

{

rt = tree[rt][\*s - tb];

if (rt == 0)

{

break;

}

}

return lv;

}

##### 左儿子右兄弟

/\*

\* 左孩子右兄弟

\* INIT: init();

\*/

const int N = 1010;

int top;

struct trie

{

char c;

int l, r, rk;

} tree[N];

void init()

{

top = 1;

memset(tree, 0, sizeof(tree[0]));

}

int sear(char \*s) // 失败返回0

{

int rt;

for (rt = 0; \*s; ++s)

{

for (rt = tree[rt].l; rt; rt = tree[rt].r)

{

if (tree[rt].c == \*s)

{

break;

}

}

if (rt == 0)

{

return 0;

}

}

return tree[rt].rk;

}

void insert(char \*s, int rk = 1) // rk: 权或者标记

{

int i, rt;

for (rt = 0; \*s; ++s, rt = i)

{

for (i = tree[rt].l; i; i = tree[i].r)

{

if (tree[i].c == \*s)

{

break;

}

}

if (i == 0)

{

tree[top].r = tree[rt].l;

tree[top].l = 0;

tree[top].c = \*s;

tree[top].rk = 0;

tree[rt].l = top;

i = top++;

}

}

tree[rt].rk = rk;

return ;

}

void delt(char \*s) // 假定s已经存在,只做标记

{

int rt;

for (rt = 0; \*s; ++s)

{

for (rt = tree[rt].l; rt; rt = tree[rt].r)

{

if (tree[rt].c == \*s)

{

break;

}

}

tree[rt].rk = 0;

}

return ;

}

int profix(char \*s) // 最长前缀

{

int rt = 0, lv;

for (lv = 0; \*s; ++s, ++lv)

{

for (rt = tree[rt].l; rt; rt = tree[rt].r)

{

if (tree[rt].c == \*s)

{

break;

}

}

if (rt == 0)

{

break;

}

}

return lv;

}

### RMQ

##### 一维

/\*

\* 求最大值,数组下标从1开始。

\* 求最小值,或者最大最小值下标,或者数组从0开始对应修改即可。

\*/

const int MAXN = 50010;

int dp[MAXN][20];

int mm[MAXN];

// 初始化RMQ,b数组下标从1开始,b数组是区间元素序列

void initRMQ(int n, int b[])

{

mm[0] = -1;

for (int i = 1; i <= n; i++)

{

mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

dp[i][0] = b[i];

}

for (int j = 1; j <= mm[n]; j++)

{

for (int i = 1; i + (1 << j) - 1 <= n; i++)

{

dp[i][j] = max(dp[i][j - 1], dp[i + (1 << (j - 1))][j - 1]);

}

}

}

// 查询最大值

int rmq(int x, int y)

{

int k = mm[y - x + 1];

return max(dp[x][k], dp[y - (1 << k) + 1][k]);

}

##### 二维

/\*

\* 二维RMQ,预处理复杂度 n\*m\*log\*(n)\*log(m)

\* 数组下标从1开始

\*/

int val[310][310];

int dp[310][310][9][9]; // 最大值

int mm[310]; // 二进制位数减一,使用前初始化

void initRMQ(int n, int m)

{

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= m; j++)

{

dp[i][j][0][0] = val[i][j];

}

}

for (int ii = 0; ii <= mm[n]; ii++)

{

for (int jj = 0; jj <= mm[m]; jj++)

{

if (ii + jj)

{

for (int i = 1; i + (1 << ii) - 1 <= n; i++)

{

for(int j = 1; j + (1 << jj) - 1 <= m; j++)

{

if (ii)

{

dp[i][j][ii][jj] = max(dp[i][j][ii - 1][jj], dp[i + (1 << (ii - 1))][j][ii - 1][jj]);

}

else

{

dp[i][j][ii][jj] = max(dp[i][j][ii][jj - 1], dp[i][j + (1 << (jj - 1))][ii][jj - 1]);

}

}

}

}

}

}

}

// 查询矩形内的最大值(x1<=x2,y1<=y2)

int rmq(int x1, int y1, int x2, int y2)

{

int k1 = mm[x2 - x1 + 1];

int k2 = mm[y2 - y1 + 1];

x2 = x2 - (1 << k1) + 1;

y2 = y2 - (1 << k2) + 1;

return max(max(dp[x1][y1][k1][k2], dp[x1][y2][k1][k2]), max(dp[x2][y1][k1][k2], dp[x2][y2][k1][k2]));

}

int main()

{

// 在外面对mm数组进行初始化

mm[0] = -1;

for (int i = 1; i <= 305; i++)

{

mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];

}

int n, m;

int Q;

int r1, c1, r2, c2;

while (scanf("%d%d", &n, &m) == 2)

{

for (int i = 1; i <= n; i++)

{

for (int j = 1; j <= m; j++)

{

scanf("%d", &val[i][j]);

}

}

initRMQ(n, m);

scanf("%d", &Q);

while(Q--)

{

scanf("%d%d%d%d", &r1, &c1, &r2, &c2);

if (r1 > r2)

{

swap(r1, r2);

}

if (c1 > c2)

{

swap(c1, c2);

}

int tmp = rmq(r1, c1, r2, c2);

printf("%d ", tmp);

if (tmp == val[r1][c1] || tmp == val[r1][c2] || tmp == val[r2][c1] || tmp == val[r2][c2])

{

printf("yes\n");

}

else

{

printf("no\n");

}

}

}

return 0;

}

### 树链部分

##### 点权

/\*

\* 基于点权,查询单点值,修改路径的上的点权

\*/

const int MAXN = 50010;

struct Edge

{

int to, next;

} edge[MAXN \* 2];

int head[MAXN], tot;

int top[MAXN]; // top[v]表示v所在的重链的顶端节点

int fa[MAXN]; // 父亲节点

int deep[MAXN]; // 深度

int num[MAXN]; // num[v]表示以v为根的子树的节点数

int p[MAXN]; // p[v]表示v对应的位置

int fp[MAXN]; // fp和p数组相反

int son[MAXN]; // 重儿子

int pos;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

pos = 1; // 使用树状数组,编号从头1开始

memset(son, -1, sizeof(son));

return ;

}

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

void dfs1(int u, int pre, int d)

{

deep[u] = d;

fa[u] = pre;

num[u] = 1;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v != pre)

{

dfs1(v, u, d + 1);

num[u] += num[v];

if (son[u] == -1 || num[v] > num[son[u]])

{

son[u] = v;

}

}

}

return ;

}

void getpos(int u, int sp)

{

top[u] = sp;

p[u] = pos++;

fp[p[u]] = u;

if (son[u] == -1)

{

return ;

}

getpos(son[u], sp);

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v != son[u] && v != fa[u])

{

getpos(v, v);

}

}

return ;

}

//树状数组

int lowbit(int x)

{

return x & (-x);

}

int c[MAXN];

int n;

int sum(int i)

{

int s = 0;

while (i > 0)

{

s += c[i];

i -= lowbit(i);

}

return s;

}

void add(int i, int val)

{

while (i <= n)

{

c[i] += val;

i += lowbit(i);

}

return ;

}

void Change(int u, int v, int val) // u->v的路径上点的值改变val

{

int f1 = top[u], f2 = top[v];

while (f1 != f2)

{

if (deep[f1] < deep[f2])

{

swap(f1, f2);

swap(u, v);

}

add(p[f1], val);

add(p[u] + 1, -val);

u = fa[f1];

f1 = top[u];

}

if (deep[u] > deep[v])

{

swap(u, v);

}

add(p[u], val);

add(p[v] + 1, -val);

return ;

}

int a[MAXN];

int main()

{

int M, P;

while (scanf("%d%d%d", &n, &M, &P) == 3)

{

int u, v;

int C1, C2, K;

char op[10];

init();

for (int i = 1; i <= n; i++)

{

scanf("%d", &a[i]);

}

while(M--)

{

scanf("%d%d", &u, &v);

addedge(u, v);

addedge(v, u);

}

dfs1(1, 0, 0);

getpos(1, 1);

memset(c, 0, sizeof(c));

for (int i = 1; i <= n; i++)

{

add(p[i],a[i]);

add(p[i] + 1, -a[i]);

}

while (P--)

{

scanf("%s", op);

if (op[0] == 'Q')

{

scanf("%d", &u);

printf("%d\n", sum(p[u]));

}

else

{

scanf("%d%d%d", &C1, &C2, &K);

if (op[0] == 'D')

{

K = -K;

}

Change(C1, C2, K);

}

}

}

return 0;

}

##### 边权

/\*

\* 基于边权,修改单条边权,查询路径边权最大值

\*/

const int MAXN = 10010;

struct Edge

{

int to, next;

} edge[MAXN \* 2];

int head[MAXN], tot;

int top[MAXN]; // top[v]表示v所在的重链的顶端节点

int fa[MAXN]; // 父亲节点

int deep[MAXN]; // 深度

int num[MAXN]; // num[v]表示以v为根的子树的节点数

int p[MAXN]; // p[v]表示v与其父亲节点的连边在线段树中的位置

int fp[MAXN]; // 和p数组相反

int son[MAXN]; // 重儿子

int pos;

void init()

{

tot = 0;

memset(head, -1, sizeof(head));

pos = 0;

memset(son, -1, sizeof(son));

return ;

}

void addedge(int u, int v)

{

edge[tot].to = v;

edge[tot].next = head[u];

head[u] = tot++;

return ;

}

void dfs1(int u, int pre, int d) // 第一遍dfs求出fa,deep,num,son

{

deep[u] = d;

fa[u] = pre;

num[u] = 1;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v != pre)

{

dfs1(v, u, d + 1);

num[u] += num[v];

if (son[u] == -1 || num[v] > num[son[u]])

{

son[u] = v;

}

}

}

}

void getpos(int u,int sp) // 第二遍dfs求出top和p

{

top[u] = sp;

p[u] = pos++;

fp[p[u]] = u;

if (son[u] == -1)

{

return ;

}

getpos(son[u], sp);

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].to;

if (v != son[u] && v != fa[u])

{

getpos(v,v);

}

}

return ;

}

// 线段树

struct Node

{

int l, r;

int Max;

} segTree[MAXN \* 3];

void build(int i, int l, int r)

{

segTree[i].l = l;

segTree[i].r = r;

segTree[i].Max = 0;

if (l == r)

{

return ;

}

int mid = (l + r) / 2;

build(i << 1, l, mid);

build((i << 1) | 1, mid + 1, r);

return ;

}

void push\_up(int i)

{

segTree[i].Max = max(segTree[i << 1].Max, segTree[(i << 1)|1].Max);

}

void update(int i, int k, int val) // 更新线段树的第k个值为val

{

if (segTree[i].l == k && segTree[i].r == k)

{

segTree[i].Max = val;

return ;

}

int mid = (segTree[i].l + segTree[i].r) / 2;

if (k <= mid)

{

update(i << 1, k, val);

}

else

{

update((i << 1) | 1, k, val);

}

push\_up(i);

return ;

}

int query(int i, int l, int r) // 查询线段树中[l,r]的最大值

{

if (segTree[i].l == l && segTree[i].r == r)

{

return segTree[i].Max;

}

int mid = (segTree[i].l + segTree[i].r) / 2;

if (r <= mid)

{

return query(i << 1, l, r);

}

else if (l > mid)

{

return query((i << 1) | 1, l, r);

}

else

{

return max(query(i << 1, l, mid), query((i << 1) | 1, mid + 1, r));

}

}

int find(int u,int v) // 查询u->v边的最大值

{

int f1 = top[u], f2 = top[v];

int tmp = 0;

while (f1 != f2)

{

if (deep[f1] < deep[f2])

{

swap(f1, f2);

swap(u, v);

}

tmp = max(tmp, query(1, p[f1], p[u]));

u = fa[f1];

f1 = top[u];

}

if (u == v)

{

return tmp;

}

if (deep[u] > deep[v])

{

swap(u, v);

}

return max(tmp, query(1, p[son[u]], p[v]));

}

int e[MAXN][3];

int main()

{

// freopen("in.txt", "r", stdin);

// freopen("out.txt", "w", stdout);

int T;

int n;

scanf("%d", &T);

while (T--)

{

init();

scanf("%d", &n);

for (int i = 0; i < n - 1; i++)

{

scanf("%d%d%d", &e[i][0], &e[i][1], &e[i][2]);

addedge(e[i][0], e[i][1]);

addedge(e[i][1], e[i][0]);

}

dfs1(1, 0, 0);

getpos(1, 1);

build(1, 0, pos - 1);

for (int i = 0; i < n-1; i++)

{

if (deep[e[i][0]] > deep[e[i][1]])

{

swap(e[i][0],e[i][1]);

}

update(1, p[e[i][1]], e[i][2]);

}

char op[10];

int u, v;

while (scanf("%s", op) == 1)

{

if (op[0] == 'D')

{

break;

}

scanf("%d%d", &u, &v);

if (op[0] == 'Q')

{

printf("%d\n", find(u, v)); // 查询u->v路径上边权的最大值

}

else

{

update(1, p[e[u - 1][1]], v); // 修改第u条边的长度为v

}

}

}

return 0;

}

### 二分查找

##### 查找V

/\*

\* 在[l, h)范围内查找值v,返回下标

\* 假设a数组已经按从小到大排序

\* 失败返回-1

\*/

int bs(int a[], int l, int h, int v)

{

int m;

while (l < h)

{

m = (l + h) >> 1;

if (a[m] == v)

{

return m;

}

if (a[m] < v)

{

l = m + 1;

}

else

{

h = m;

}

}

return -1;

}

##### 查找大于等于v的第一个值

/\*

\* 传入参数必须在a[l]与a[h]之间

\* 假设a数组已经按从小到大排序

\* 返回值l总是合理的

\*/

int bs(int a[], int l, int h, int v)

{

int m;

while (l < h)

{

m = (l + h) >> 1;

if (a[m] < v)

{

l = m + 1;

}

else

{

h = m;

}

}

return l;

}

##### 查找小于等于v的最后一个值

/\*

\* 在下标[l, r]范围内查找,返回下标

\* 假设a数组已经按从小到大排序

\* 失败返回-1

\*/

int bs(int a[], int l, int r, int v)

{

int m;

while (l < r)

{

m = (l + r + 1) >> 1;

if (a[m] > v)

{

r = m - 1;

}

else

{

l = m;

}

}

if (a[l] > v)

{

return -1;

}

return l;

}

##### 二分套二分

/\*

\* 二分套二分

\* 数组A同数组B组合乘积，二分查找第K大

\*/

typedef long long ll;

const int MAXN = 5e4 + 10;

ll N, K;

ll A[MAXN];

ll B[MAXN];

// 查找小于x的元素个数

ll check(ll x)

{

ll j = N, ans = 0;

for (int i = 1; i <= N; i++)

{

for (; j > 0;)

{

if (A[i] \* B[j] > x)

{

j--;

}

else

{

break;

}

}

ans += j;

}

return ans;

}

int main(int argc, const char \* argv[])

{

cin >> N >> K;

for (int i = 1; i <= N; i++)

{

scanf("%lld %lld", A + i, B + i);

}

sort(A + 1, A + N + 1);

sort(B + 1, B + N + 1);

ll ans = 0;

ll key = N \* N - K + 1;

ll low = A[1] \* B[1]; // 初始最小值

ll high = A[N] \* B[N]; // 初始最大值

while (high - low > 1)

{

ll mid = (low + high) >> 1;

if (check(mid) >= key)

{

ans = mid;

high = mid;

}

else

{

low = mid;

}

}

cout << ans << '\n';

return 0;

}

### 树状数组

/\*

单点更新，区间查询，（也可区间更新加），逆序对等等

\*/

int lowerbit(int x) {

return x & -x;

}

void add(int p, int x) {

while (p < maxn) {

d[p] += x;

p += lowerbit(p);

}

}

int sum(int p) {

int res = 0;

while (p) {

res += d[p];

p -= lowerbit(p);

}

return res;

}

#include<iostream>

#include<cstdio>

#include<cstring>

#include<string>

#include<cmath>

using namespace std;

typedef long long ll;

const int maxn = 50005;

int a[maxn];

int n;

int lowbit(const int t) {

return t & (-t);

}

void insert(int t, int d) {

while (t <= n){

a[t] += d;

t = t + lowbit(t);

}

}

ll getSum(int t) {

ll sum = 0;

while (t > 0){

sum += a[t];

t = t - lowbit(t);

}

return sum;

}

int main() {

int t, k, d;

scanf("%d", &t);

k= 1;

while (t--){

memset(a, 0, sizeof(a));

scanf("%d", &n);

for (int i = 1; i <= n; ++i) {

scanf("%d", &d);

insert(i, d);

}

string str;

printf("Case %d:\n", k++);

while (cin >> str) {

if (str == "End") break;

int x, y;

scanf("%d %d", &x, &y);

if (str == "Query")

printf("%lld\n", getSum(y) - getSum(x - 1));

else if (str == "Add")

insert(x, y);

else if (str == "Sub")

insert(x, -y);

}

}

return 0;

}

##### 一维

/\*

\* INIT: ar[]置为0;

\* CALL: add(i, v): 将i点的值加v; sum(i): 求[1, i]的和;

\*/

#define typev int // type of res

const int N = 1010;

typev ar[N]; // index: 1 ~ N

int lowb(int t)

{

return t & (-t);

}

void add(int i, typev v)

{

for (; i < N; ar[i] += v, i += lowb(i));

return ;

}

typev sum(int i)

{

typev s = 0;

for (; i > 0; s += ar[i], i -= lowb(i));

return s;

}

##### 二维

/\*

\* INIT: c[][]置为0; Row,Col要赋初值

\*/

const int N = 10000;

int c[N][N];

int Row, Col;

inline int Lowbit(const int &x)

{ // x > 0

return x & (-x);

}

int Sum(int i, int j)

{

int tempj, sum = 0;

while (i > 0)

{

tempj = j;

while (tempj > 0)

{

sum += c[i][tempj];

tempj -= Lowbit(tempj);

}

i -= Lowbit(i);

}

return sum;

}

void Update(int i, int j, int num)

{

int tempj;

while (i <= Row)

{

tempj = j;

while (tempj <= Col)

{

c[i][tempj] += num;

tempj += Lowbit(tempj);

}

i += Lowbit(i);

}

return ;

}

### 滚动数组

##### 一维

int main()

{

int i;

long long d[80];

d[0] = 1;

d[1] = 1;

for(i = 2; i < 80; i++)

{

d[i] = d[i - 1] + d[i - 2];

}

printf("%lld\n",d[79]);

return 0;

}

上面这个循环d[i]只依赖于前两个数据d[i - 1]和d[i - 2]; 为了节约空间用滚动数组的做法。

int main()

{

int i;

long long d[3];

d[0] = 1;

d[1] = 1;

for(i = 2; i < 80; i++)

{

d[i % 3] = d[(i - 1) % 3] + d[(i - 2) % 3];

}

printf("%lld\n", d[79%3]);

return 0;

}

上面的取余运算，我们成功地只保留了需要的最后3个解，数组好象在“滚动”一样，所以叫滚动数组(对于二维也可以用)。所以，很明显，滚动数组可以通过取余（％）来实现的，但是这里存在一个通病，那就是时间换内存一定会牺牲时间。因此，滚动数组一般用在时间比较充裕，而内存不够的情况下。

##### 二维

int i, j, d[100][100];

for(i = 1; i < 100; i++)

for(j = 0; j < 100; j++)

d[i][j] = d[i - 1][j] + d[i][j - 1];

上面的d[i][j]只依赖于d[i - 1][j], d[i][j - 1];   
运用滚动数组

int i, j, d[2][100];

for(i = 1; i < 100; i++)

for(j = 0; j < 100; j++)

d[i % 2][j] = d[(i - 1) % 2][j] + d[i % 2][j - 1];

##### 附录

附上另外一种滚动数组的写法（其实和取余一个路子）：   
斐波那契数列……

int Fib[3];

int fib(int n)

{

Fib[1] = 0;

Fib[2] = 1;

for(int i = 2; i <= n; ++i)

{

Fib[0] = Fib[1];

Fib[1] = Fib[2];

Fib[2] = Fib[0] + Fib[1];

}

return Fib[2];

}

int main()

{

int ncase, n, ans;

scanf("%d", &ncase);

while(ncase--)

{

scanf("%d", &n);

ans = fib(n);

printf("%d\n", ans);

}

return 0;

}

### 逆序数

##### 归并排序求逆序数

/\*

\* 也可以用树状数组做

\* a[0...n-1] cnt=0; call: MergeSort(0, n)

\*/

const int N = 1010;

int a[N];

int c[N];

int cnt = 0;

void MergeSort(int l, int r)

{

int mid, i, j, tmp;

if (r > l + 1)

{

mid = (l + r) / 2;

MergeSort(l, mid);

MergeSort(mid, r);

tmp = l;

for (i = l, j = mid; i < mid && j < r;)

{

if (a[i] > a[j])

{

c[tmp++] = a[j++];

cnt += mid - i;

}

else

{

c[tmp++] = a[i++];

}

}

if (j < r)

{

for (; j < r; ++j)

{

c[tmp++] = a[j];

}

}

else

{

for (; i < mid; ++i)

{

c[tmp++]=a[i];

}

}

for (i = l; i < r; ++i)

{

a[i] = c[i];

}

}

return ;

}

##### 逆序数推排列数

/\*

\* 动态规划:f(n,m)表示逆序数为m的n元排列的个数,则

\* f(n + 1, m) = f(n, m) + f(n, m - 1) + ... + f(n, m - n)(当 b<0 时,f(a,b) = 0)

\* 优化又考虑到如果直接利用上式计算时间复杂度为O(n^3),我们分析上

\* 式不难发现f(n + 1, m) = f(n, m) + f(n + 1, m - 1)

\* end = node[index].e,

\* if(m-n-1 >= 0) f(n+1, m) -= f(n, m-n-1).

\*/

const int N = 1001;

const int C = 10001;

const long MOD = 1000000007;

long arr[N][C];

long long temp;

int main()

{

int i, j;

arr[1][0] = arr[2][0] = arr[2][1] = 1;

for (i = 3; i < N; ++i)

{

arr[i][0] = 1;

long h = i \* (i + 1) / 2 + 1;

if (h > C)

{

h = C;

}

for (j = 1; j < h; ++j)

{

temp = arr[i - 1][j] + arr[i][j - 1];

arr[i][j] = temp % MOD;

if (j - i >= 0)

{

arr[i][j] -= arr[i - 1][j - i];

if (arr[i][j] < 0)

{ // 注意:由于arr[i][j]和arr[i - 1][j - i]都是模过的,所以可能会得到负数

arr[i][j] += MOD;

}

}

}

}

while (scanf("%d %d", &i, &j) != EOF)

{

printf("%ld\n", arr[i][j]);

}

return 0;

}

### 并查集

/\*

\* INIT: makeset(n);

\* CALL: findset(x); unin(x, y);

\*/

const int N = 1010;

struct lset

{

int p[N], rank[N], sz;

void link(int x, int y)

{

if (x == y)

return ;

if (rank[x] > rank[y])

p[y] = x;

else

p[x] = y;

if (rank[x] == rank[y])

rank[y]++;

return ;

}

void makeset(int n)

{

sz = n;

for (int i = 0; i < sz; i++)

{

p[i] = i;

rank[i] = 0;

}

return ;

}

int findset(int x)

{

if (x != p[x])

p[x] = findset(p[x]);

return p[x];

}

void unin(int x, int y)

{

link(findset(x), findset(y));

return ;

}

void compress()

{

for (int i = 0; i < sz; i++)

findset(i);

return ;

}

};

### 快速排序

void ksort(int l, int h, int a[])

{

if (h < l + 2)

{

return ;

}

int e = h, p = l;

while (l < h)

{

while (++l < e && a[l] <= a[p]);

while (--h > p && a[h] >= a[p]);

if (l < h)

{

swap(a[l], a[h]);

}

}

swap(a[h], a[p]);

ksort(p, h, a);

ksort(l, e, a);

return ;

}

### 机器工作调度

2台机器,n件任务,必须先在S1上做,再在S2上做.

任务之间先做后做任意.求最早的完工时间.

这是一个经典问题: 2台机器的情况下有多项式算法(Johnson算法),3台或以上的机器是NP-hard算法。

Johnson算法:

(1)把作业按工序加工时间分成两个子集,第一个集合中在S1上做的时间比在S2上少,其它的作业放到第二个集合;

　 先完成第一个集合里面的作业,再完成第二个集合里的作业.

(2)对于第一个集合,其中的作业顺序是按在S1上的时间的不减排列;

　 对于第二个集合, 其中的作业顺序是按在S2上的时间的不增排列.

const int MAXN = 5e4 + 5;

struct task

{

int a;

int b;

} TaskA[MAXN], TaskB[MAXN];

bool cmpA(task a, task b)

{

return a.a <= b.a;

}

bool cmpB(task a, task b)

{

return a.b >= b.b;

}

int main(int argc, const char \* argv[])

{

int N;

cin >> N;

int a, b;

int posA = 0, posB = 0;

int sumA = 0, sumB = 0;

for (int i = 0; i < N; i++)

{

scanf("%d %d", &a, &b);

if (a < b)

{

TaskA[posA].a = a;

TaskA[posA++].b = b;

sumA += b;

}

else

{

TaskB[posB].a = a;

TaskB[posB++].b = b;

sumB += a;

}

}

sort(TaskA, TaskA + posA, cmpA);

sort(TaskB, TaskB + posB, cmpB);

for (int i = 0; i < posB; i++)

{

TaskA[posA++] = TaskB[i];

}

int ans = TaskA[0].a + TaskA[0].b;

int sum = TaskA[0].a;

for (int i = 1; i < posA; i++)

{

sum += TaskA[i].a;

ans = sum < ans ? ans + TaskA[i].b : sum + TaskA[i].b;

}

cout << ans << '\n';

return 0;

}

### 取第k个元素

/\*

\* 取第k个元素

\* k = 0 ... n - 1,平均复杂度O(n) 注意a[]中的顺序被改变

\*/

#define \_cp(a,b) ((a) < (b))

typedef int elem\_t;

elem\_t kth\_element(int n, elem\_t \*a, int k)

{ // a[0 ... n-1]

elem\_t t, key;

int l = 0, r = n - 1, i, j;

while (l < r)

{

for (key = a[((i = l - 1) + (j = r + 1)) >> 1]; i < j;)

{

for (j--; \_cp(key, a[j]); j--);

for (i++; \_cp(a[i], key); i++);

if (i < j)

{

t = a[i], a[i] = a[j], a[j] = t;

}

}

if (k > j)

{

l = j + 1;

}

else

{

r = j;

}

}

return a[k];

}

### 最长公共递增子序列

/\*

\* 最长公共递增子序列 O(n^2)

\* f记录路径,DP记录长度, 用a对b扫描,逐步最优化。

\*/

const int N = 1010;

int f[N][N], dp[N];

int gcis(int a[], int la, int b[], int lb, int ans[])

{ // a[1...la], b[1...lb]

int i, j, k, mx;

memset(f, 0, sizeof(f));

memset(dp, 0, sizeof(dp));

for (i = 1; i <= la; i++)

{

memcpy(f[i], f[i-1], sizeof(f[0]));

for (k = 0, j = 1; j <= lb; j++)

{

if (b[j - 1] < a[i - 1] && dp[j] > dp[k])

{

k = j;

}

if (b[j - 1] == a[i - 1] && dp[k] + 1 > dp[j])

{

dp[j] = dp[k] + 1,

f[i][j] = i \* (lb + 1) + k;

}

}

}

for (mx = 0, i = 1; i <= lb; i++)

{

if (dp[i] > dp[mx])

{

mx = i;

}

}

for (i = la \* lb + la + mx, j = dp[mx]; j; i = f[i / (lb + 1)][i % (lb + 1)], j--)

{

ans[j - 1] = b[i % (lb + 1) - 1];

}

return dp[mx];

}

### 最长公共子序列

const int N = 1010;

int a[N][N];

int LCS(const char \*s1, const char \*s2)

{ // s1:0...m, s2:0...n

int m = (int)strlen(s1), n = (int)strlen(s2);

int i, j;

a[0][0] = 0;

for (i = 1; i <= m; ++i)

a[i][0] = 0;

for (i = 1; i <= n; ++i)

a[0][i] = 0;

for (i = 1; i <= m; ++i)

{

for (j = 1; j <= n; ++j)

{

if (s1[i - 1] == s2[j - 1])

a[i][j] = a[i - 1][j - 1] + 1;

else if (a[i - 1][j] > a[i][j - 1])

a[i][j]= a[i - 1][j];

else

a[i][j] = a[i][j - 1];

}

}

return a[m][n];

}

### 最少找硬币问题

/\*

\* 贪心策略-深度搜索

\*/

int value[7] = {100, 50, 20, 10, 5, 2, 1};

int count[7]; // count[i]:value[i]硬币的个数

int res[7];

bool flag;

void DFS(int total, int p);

int main()

{

int pay = 0;

scanf("%d", &pay);

//...

flag = false; // 标识是否已经找到结果

for (int i = 0; i < 7; ++i)

{

res[i] = 0;

}

DFS(pay, 0); // pay为要找的钱数

if (flag)

{

printf("Accept\n%d", res[0]);

for (int i = 1; i < 7; ++i)

{

printf(" %d", res[i]);

}

printf("\n");

}

else

{

printf("Refuse\n"); // 无法正好找钱

}

}

void DFS(int total, int p)

{

if (flag)

{

return ;

}

if (p == 7)

{

if (total == 0)

{

flag = true;

}

return ;

}

int i, max = total / value[p];

if (max > count[p])

{

max = count[p];

}

for (i = max; i >= 0; --i)

{

res[p] = i;

DFS(total - i \* value[p], p + 1);

if (flag)

return ;

}

return ;

}

### 棋盘分割

/\*

\* 棋盘分割

\* 将一个8\*8的棋盘进行如下分割:将原棋盘割下一块矩形棋盘并使剩下部

\* 分也是矩形,再将剩下的部分继续如此分割,这样割了(n-1)次后,连同最

\* 后剩下的矩形棋盘共有n块矩形棋盘。(每次切割都只能沿着棋盘格子的边

\* 进行) 原棋盘上每一格有一个分值,一块矩形棋盘的总分为其所含各格分

\* 值之和。现在需要把棋盘按上述规则分割成n块矩形棋盘,并使各矩形棋

\* 盘总分的均方差最小。 均方差...,其中平均值...,xi为第i块矩形棋盘的

\* 总分。请编程对给出的棋盘及 n,求出 O'的最小值。

\*/

#define min(a, b) ((a) < (b) ? (a) : (b))

const int oo = 10000000;

int map[8][8];

double C[16][8][8][8][8]; // c[k][si][ei][sj][ej]: 对矩阵

// map[si...sj][ei...ej]分割成k个矩形(切割k-1刀)的结果

double ans; // 平均值

int n; // 分成n块矩形棋盘

void input(void);

void reset(void);

double caluate(int i1, int j1, int i2, int j2);

void dp(int m, int si, int sj, int ei, int ej);

int main()

{

int m, i, j, k, l;

while (scanf("%d", &n) != EOF)

{

input();

reset();

for (m = 1; m <= n; m++)

{

for (i = 0; i < 8; i++)

{

for (j = 0; j < 8; j++)

{

for (k = 0; k < 8; k++)

{

for (l = 0; l < 8; l++)

{

if ((k - i + 1) \* (l - j + 1) < m)

{

C[m][i][j][k][l] = oo;

}

else

{

if (m == 1)

{

C[m][i][j][k][l] = pow((caluate(i, j, k, l) - ans), 2);

}

else

{

dp(m, i, j, k, l);

}

}

}

}

}

}

}

printf("%.3lf\n", sqrt(C[n][0][0][7][7] / n));

}

return 0;

}

void input()

{

int i, j;

double sum = 0;

for (i = 0; i < 8; i++)

{

for (j = 0; j < 8; j++)

{

scanf("%d", &map[i][j]);

sum += map[i][j];

}

}

ans = sum / double(n); // 平均值

}

void reset()

{

int i, j, k, l, m;

for (m = 0; m <= n; m++)

{

for (i = 0; i < 8; i++)

{

for (j = 0; j < 8; j++)

{

for (k = 0; k < 8; k++)

{

for (l = 0; l < 8; l++)

{

C[m][i][j][k][l] = 0;

}

}

}

}

}

return ;

}

double caluate(int i1, int j1, int i2, int j2)

{

double sum = 0;

int i, j;

for (i = i1; i <= i2; i++)

{

for (j = j1; j <= j2; j++)

{

sum += map[i][j];

}

}

return sum;

}

void dp(int m, int si, int sj, int ei, int ej)

{

int i, j;

double mins = oo;

for (j = sj; j < ej; j++)

{ // 竖刀

mins = min(mins, C[1][si][sj][ei][j] + C[m - 1][si][j + 1][ei][ej]);

mins = min(mins, C[m - 1][si][sj][ei][j] + C[1][si][j + 1][ei][ej]);

}

for (i = si; i < ei; i++)

{ // 横刀

mins = min(mins, C[1][si][sj][i][ej] + C[m - 1][i + 1][sj][ei][ej]);

mins = min(mins, C[m - 1][si][sj][i][ej] + C[1][i + 1][sj][ei][ej]);

}

C[m][si][sj][ei][ej] = mins;

return ;

}

### 堆栈

const int MAXSIZE = 10000;

int a[MAXSIZE], heapsize;

inline void swap(int i, int j)

{

int temp = a[i];

a[i] = a[j];

a[j] = temp;

return ;

}

inline int Parent(int i)

{

return i >> 1;

}

inline int Left(int i)

{

return 1 << i;

}

inline int Right(int i)

{

return (1 << i) + 1; // 保持堆的性质

}

void MaxHeapify(int i)

{

int l = Left(i), r = Right(i), largest;

if (l <= heapsize && a[l] > a[i])

{

largest = l;

}

else

{

largest = i;

}

if (r <= heapsize && a[r] > a[largest])

{

largest = r;

}

if (largest != i)

{

swap(i, largest);

MaxHeapify(largest);

}

return ;

}

void BuildMaxHeap(int \*arr, int n)

{

heapsize = n;

for (int i = heapsize / 2; i > 0; --i)

{

MaxHeapify(i);

}

return ;

}

void HeapSort(int \*arr, int n)

{

BuildMaxHeap(arr, n);

for (int i = n; i > 1; --i)

{

swap(1, i);

heapsize--;

MaxHeapify(1);

}

return ;

}

### 背包相关

const int MAXN = 101;

const int SIZE = 50001;

int dp[SIZE];

int volume[MAXN], value[MAXN], c[MAXN];

int n, v; // 总物品数，背包容量

// 01背包

void ZeroOnepark(int val, int vol)

{

for (int j = v ; j >= vol; j--)

{

dp[j] = max(dp[j], dp[j - vol] + val);

}

}

// 完全背包

void Completepark(int val, int vol)

{

for (int j = vol; j <= v; j++)

{

dp[j] = max(dp[j], dp[j - vol] + val);

}

}

// 多重背包

void Multiplepark(int val, int vol, int amount)

{

if (vol \* amount >= v)

{

Completepark(val, vol);

}

else

{

int k = 1;

while (k < amount)

{

ZeroOnepark(k \* val, k \* vol);

amount -= k;

k <<= 1;

}

if (amount > 0)

{

ZeroOnepark(amount \* val, amount \* vol);

}

}

}

int main()

{

while (cin >> n >> v)

{

for (int i = 1 ; i <= n ; i++)

{

cin >> volume[i] >> value[i] >> c[i]; // 费用，价值，数量

}

memset(dp, 0, sizeof(dp));

for (int i = 1; i <= n; i++)

{

Multiplepark(value[i], volume[i], c[i]);

}

cout << dp[v] << endl;

}

return 0;

}

### 使序列有序的最少交换次数

##### 交换相邻两数

如果只是交换相邻两数，那么最少交换次数为该序列的逆序数。

##### 交换任意两数

/\*

\* 交换任意两数的本质是改变了元素位置，

\* 故建立元素与其目标状态应放置位置的映射关系

\*/

int getMinSwaps(vector<int> &A)

{

// 排序

vector<int> B(A);

sort(B.begin(), B.end());

map<int, int> m;

int len = (int)A.size();

for (int i = 0; i < len; i++)

{

m[B[i]] = i; // 建立每个元素与其应放位置的映射关系

}

int loops = 0; // 循环节个数

vector<bool> flag(len, false);

// 找出循环节的个数

for (int i = 0; i < len; i++)

{

if (!flag[i])

{

int j = i;

while (!flag[j])

{

flag[j] = true;

j = m[A[j]]; // 原序列中j位置的元素在有序序列中的位置

}

loops++;

}

}

return len - loops;

}

vector<int> nums;

int main()

{

nums.push\_back(1);

nums.push\_back(2);

nums.push\_back(4);

nums.push\_back(3);

nums.push\_back(5);

int res = getMinSwaps(nums);

cout << res << '\n';

return 0;

}

##### 交换任意区间

/\*

\* 默认目标映射关系是 key 1 => val 1 …… key n => val n

\* 如果序列不是 1~n 可以通过 map 建立新的目标映射关系

\* 交换任意区间的本质是改变了元素的后继，故建立元素与其初始状态后继的映射关系

\*/

const int MAXN = 30;

int n;

int vis[MAXN];

int A[MAXN], B[MAXN];

int getMinSwaps()

{

memset(vis, 0, sizeof(vis));

for (int i = 1; i <= n; i++)

{

B[A[i]] = A[i % n + 1];

}

for (int i = 1; i <= n; i++)

{

B[i] = (B[i] - 2 + n) % n + 1;

}

int cnt = n;

for (int i = 1; i <= n; i++)

{

if (vis[i])

{

continue;

}

vis[i] = 1;

cnt--;

for (int j = B[i]; j != i; j = B[j])

{

vis[j] = 1;

}

}

return cnt;

}

int main()

{

cin >> n;

for (int i = 1; i <= n; i++)

{

cin >> A[i];

}

int res = getMinSwaps();

cout << res << '\n';

return 0;

}

# Geometry 计算几何

### Graham求凸包

/\*

\* Graham 求凸包 O(N \* logN)

\* CALL: nr = graham(pnt, int n, res); res[]为凸包点集;

\*/

struct point

{

double x, y;

};

bool mult(point sp, point ep, point op)

{

return (sp.x - op.x) \* (ep.y - op.y) >= (ep.x - op.x) \* (sp.y - op.y);

}

//inline bool operator < (const point &l, const point &r)

//{

// return l.y < r.y || (l.y == r.y && l.x < r.x);

//}

int graham(point pnt[], int n, point res[])

{

int i, len, top = 1;

sort(pnt, pnt + n);

if (n == 0)

{

return 0;

}

res[0] = pnt[0];

if (n == 1)

{

return 1;

}

res[1] = pnt[1];

if (n == 2)

{

return 2;

}

res[2] = pnt[2];

for (i = 2; i < n; i++)

{

while (top && mult(pnt[i], res[top], res[top - 1]))

{

top--;

}

res[++top] = pnt[i];

}

len = top;

res[++top] = pnt[n - 2];

for (i = n - 3; i >= 0; i--)

{

while (top != len && mult(pnt[i], res[top], res[top - 1]))

{

top--;

}

res[++top] = pnt[i];

}

return top; // 返回凸包中点的个数

}

### 判断四点共面

##### 混合积

struct point

{

double x, y, z;

point operator - (point &o)

{

point ans;

ans.x = this->x - o.x;

ans.y = this->y - o.y;

ans.z = this->z - o.z;

return ans;

}

};

double dot\_product(const point &a, const point &b)

{

return a.x \* b.x + a.y \* b.y + a.z \* b.z;

}

point cross\_product(const point &a, const point &b)

{

point ans;

ans.x = a.y \* b.z - a.z \* b.y;

ans.y = a.z \* b.x - a.x \* b.z;

ans.z = a.x \* b.y - a.y \* b.x;

return ans;

}

int main()

{

point p[4];

int T;

for (scanf("%d", &T); T--;)

{

for (int i = 0; i < 4; ++i)

{

scanf("%lf%lf%lf", &p[i].x, &p[i].y, &p[i].z);

}

puts(dot\_product(p[3] - p[0], cross\_product(p[2] - p[0], p[1] - p[0])) == 0.0 ? "Yes\n" : "No\n");

}

return 0;

}

### 判断线段与圆是否相交

typedef long long ll;

typedef struct // 点结构

{

ll x, y;

} Point;

Point A, B, C, O; // 三角形三点与圆心

ll r; // 半径

// 判断线段是否和圆相交

int segOnCircle(Point \*p\_1, Point \*p\_2)

{

ll a, b, c, dist\_1, dist\_2, angle\_1, angle\_2; // ax + by + c = 0;

if (p\_1->x == p\_2->x) // 当x相等

{

a = 1, b = 0, c = -p\_1->x;

}

else if (p\_1->y == p\_2->y) // 当y相等

{

a = 0, b = 1, c = -p\_1->y;

}

else

{

a = p\_1->y - p\_2->y;

b = p\_2->x - p\_1->x;

c = p\_1->x \* p\_2->y - p\_1->y \* p\_2->x;

}

dist\_1 = a \* O.x + b \* O.y + c;

dist\_1 \*= dist\_1;

dist\_2 = (a \* a + b \* b) \* r \* r;

if (dist\_1 > dist\_2)

{

return 0;

}

angle\_1 = (O.x - p\_1->x) \* (p\_2->x - p\_1->x) + (O.y - p\_1->y) \* (p\_2->y - p\_1->y);

angle\_2 = (O.x - p\_2->x) \* (p\_1->x - p\_2->x) + (O.y - p\_2->y) \* (p\_1->y - p\_2->y);

if (angle\_1 > 0 && angle\_2 > 0)

{

return 1;

}

return 0;

}

### 求多边形重心

/\*

\* 求多边形重心

\* INIT: pnt[]已按顺时针(或逆时针)排好序; | CALL: res = bcenter(pnt, n);

\*/

struct point

{

double x, y;

};

point bcenter(point pnt[], int n)

{

point p, s;

double tp, area = 0, tpx = 0, tpy = 0;

p.x = pnt[0].x;

p.y = pnt[0].y;

for (int i = 1; i <= n; ++i)

{ // point:0 ~ n - 1

s.x = pnt[(i == n) ? 0 : i].x;

s.y = pnt[(i == n) ? 0 : i].y;

tp = (p.x \* s.y - s.x \* p.y);

area += tp / 2;

tpx += (p.x + s.x) \* tp;

tpy += (p.y + s.y) \* tp;

p.x = s.x;

p.y = s.y;

}

s.x = tpx / (6 \* area);

s.y = tpy / (6 \* area);

return s;

}

### 三角形相关重点

##### 三角形重点

设三角形的三条边为a, b, c, 且不妨假设a <= b <= c.

##### 面积

三角形面积可以根据海伦公式求得：

s = sqrt(p \* (p - a) \* (p - b) \* (p - c));   
p = (a + b + c) / 2;

##### 关键点与A, B, C三顶点距离之和

##### 费马点

该点到三角形三个顶点的距离之和最小。

有个有趣的结论:

若三角形的三个内角均小于120度,那么该点连接三个顶点形成的三个角均为120度;若三角形存在一个内角大于120度,则该顶点就是费马点。

计算公式如下:

若有一个内角大于120度(这里假设为角C),则距离为a + b;若三个内角均小于120度,则距离为sqrt((a \* a + b \* b + c \* c + 4 \* sqrt(3.0) \* s) / 2)。

##### 内心

角平分线的交点。

令x = (a + b - c) / 2, y = (a - b + c) / 2, z = (-a + b + c) / 2, h = s / p.

计算公式为sqrt(x \* x + h \* h) + sqrt(y \* y + h \* h) + sqrt(z \* z + h \* h)。

##### 重心

中线的交点。

计算公式如下:

2.0 / 3 \* (sqrt((2 \* (a \* a + b \* b) - c \* c) / 4)

+ sqrt((2 \* (a \* a + c \* c) - b \* b) / 4) + sqrt((2 \* (b \* b + c \* c) - a \* a) / 4))。

##### 垂心

垂线的交点。   
计算公式如下:   
3 \* (c / 2 / sqrt(1 - cosC \* cosC))

##### 外心

三点求圆心坐标

Point waixin(Point a, Point b, Point c)

{

double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 \* a1 + b1 \* b1) / 2;

double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 \* a2 + b2 \* b2) / 2;

double d = a1 \* b2 - a2 \* b1;

return Point(a.x + (c1 \* b2 - c2 \* b1) / d, a.y + (a1 \* c2 -a2 \* c1) / d);

}

### 平面最近点对

/\*

\* O(N \* logN)

\*/

const int N = 100005;

const double MAX = 10e100, eps = 0.00001;

struct Point

{

double x, y;

int index;

};

Point a[N], b[N], c[N];

double closest(Point \*, Point \*, Point \*, int, int);

double dis(Point, Point);

int cmp\_x(const void \*, const void\*);

int cmp\_y(const void \*, const void\*);

int merge(Point \*, Point \*, int, int, int);

inline double min(double, double);

int main()

{

int n, i;

double d;

scanf("%d", &n);

while (n)

{

for (i = 0; i < n; i++)

{

scanf("%lf%lf", &(a[i].x), &(a[i].y));

}

qsort(a, n, sizeof(a[0]), cmp\_x);

for (i = 0; i < n; i++)

{

a[i].index = i;

}

memcpy(b, a, n \* sizeof(a[0]));

qsort(b, n, sizeof(b[0]), cmp\_y);

d = closest(a, b, c, 0, n - 1);

printf("%.2lf\n", d);

scanf("%d", &n);

}

return 0;

}

double closest(Point a[],Point b[],Point c[],int p,int q)

{

if (q - p == 1)

{

return dis(a[p], a[q]);

}

if (q - p == 2)

{

double x1 = dis(a[p], a[q]);

double x2 = dis(a[p + 1], a[q]);

double x3 = dis(a[p], a[p + 1]);

if (x1 < x2 && x1 < x3)

{

return x1;

}

else if (x2 < x3)

{

return x2;

}

else

{

return x3;

}

}

int i, j, k, m = (p + q) / 2;

double d1, d2;

for (i = p, j = p, k = m + 1; i <= q; i++)

{

if (b[i].index <= m)

{

c[j++] = b[i]; // 数组c左半部保存划分后左部的点, 且对y是有序的.

}

else

{

c[k++] = b[i];

}

}

d1 = closest(a, c, b, p, m);

d2 = closest(a, c, b, m + 1, q);

double dm = min(d1, d2);

// 数组c左右部分分别是对y坐标有序的,将其合并到b.

merge(b, c, p, m, q);

for (i = p, k = p; i <= q; i++)

{

if (fabs(b[i].x - b[m].x) < dm)

{

c[k++] = b[i]; // 找出离划分基准左右不超过dm的部分, 且仍然对y坐标有序.

}

}

for (i = p; i < k; i++)

{

for (j = i + 1; j < k && c[j].y - c[i].y < dm; j++)

{

double temp = dis(c[i], c[j]);

if (temp < dm)

{

dm = temp;

}

}

}

return dm;

}

double dis(Point p, Point q)

{

double x1 = p.x - q.x, y1 = p.y - q.y;

return sqrt(x1 \*x1 + y1 \* y1);

}

int merge(Point p[], Point q[], int s, int m, int t)

{

int i, j, k;

for (i = s, j = m + 1, k = s; i <= m && j <= t;)

{

if (q[i].y > q[j].y)

{

p[k++] = q[j], j++;

}

else

{

p[k++] = q[i], i++;

}

}

while (i <= m)

{

p[k++] = q[i++];

}

while (j <= t)

{

p[k++] = q[j++];

}

memcpy(q + s, p + s, (t - s + 1) \* sizeof(p[0]));

return 0;

}

int cmp\_x(const void \*p, const void \*q)

{

double temp = ((Point\*)p)->x - ((Point\*)q)->x;

if (temp > 0)

{

return 1;

}

else if (fabs(temp) < eps)

{

return 0;

}

else

{

return - 1;

}

}

int cmp\_y(const void \*p, const void \*q)

{

double temp = ((Point\*)p)->y - ((Point\*)q)->y;

if (temp > 0)

{

return 1;

}

else if (fabs(temp) < eps)

{

return 0;

}

else

{

return - 1;

}

}

inline double min(double p, double q)

{

return (p > q) ? (q): (p);

}

### 旋转卡壳

##### 求解平面最远点对

struct Point

{

int x, y;

Point(int \_x = 0, int \_y = 0)

{

x = \_x;

y = \_y;

}

Point operator - (const Point &b)const

{

return Point(x - b.x, y - b.y);

}

int operator ^(const Point &b)const

{

return x \* b.y - y \* b.x;

}

int operator \*(const Point &b)const

{

return x \* b.x + y \* b.y;

}

void input()

{

scanf("%d%d", &x, &y);

return ;

}

};

// 距离的平方

int dist2(Point a, Point b)

{

return (a - b) \* (a - b);

}

// 二维凸包,int

const int MAXN = 50010;

Point list[MAXN];

int Stack[MAXN], top;

bool \_cmp(Point p1, Point p2)

{

int tmp = (p1 - list[0]) ^ (p2 - list[0]);

if (tmp > 0)

{

return true;

}

else if (tmp == 0 && dist2(p1, list[0]) <= dist2(p2, list[0]))

{

return true;

}

else

{

return false;

}

}

void Graham(int n)

{

Point p0;

int k = 0;

p0 = list[0];

for (int i = 1; i < n; i++)

{

if (p0.y > list[i].y || (p0.y == list[i].y && p0.x > list[i].x))

{

p0 = list[i];

k = i;

}

}

swap(list[k], list[0]);

sort(list + 1, list + n, \_cmp);

if (n == 1)

{

top = 1;

Stack[0] = 0;

return ;

}

if (n == 2)

{

top = 2;

Stack[0] = 0;

Stack[1] = 1;

return ;

}

Stack[0] = 0;

Stack[1] = 1;

top = 2;

for (int i = 2; i < n; i++)

{

while (top > 1 && ((list[Stack[top - 1]] - list[Stack[top - 2]]) ^ (list[i] - list[Stack[top - 2]])) <= 0)

{

top--;

}

Stack[top++] = i;

}

return ;

}

// 旋转卡壳,求两点间距离平方的最大值

int rotating\_calipers(Point p[],int n)

{

int ans = 0;

Point v;

int cur = 1;

for (int i = 0; i < n; i++)

{

v = p[i] - p[(i + 1) % n];

while ((v ^ (p[(cur + 1) % n] - p[cur])) < 0)

{

cur = (cur + 1) % n;

}

ans = max(ans, max(dist2(p[i], p[cur]), dist2(p[(i + 1) % n], p[(cur + 1) % n])));

}

return ans;

}

Point p[MAXN];

int main()

{

int n;

while (scanf("%d", &n) == 1)

{

for (int i = 0; i < n; i++)

{

list[i].input();

}

Graham(n);

for (int i = 0; i < top; i++)

{

p[i] = list[Stack[i]];

}

printf("%d\n", rotating\_calipers(p, top));

}

return 0;

}

##### 求解平面点集最大三角形

struct Point

{

int x, y;

Point(int \_x = 0, int \_y = 0)

{

x = \_x;

y = \_y;

}

Point operator - (const Point &b)const

{

return Point(x - b.x, y - b.y);

}

int operator ^(const Point &b)const

{

return x \* b.y - y \* b.x;

}

int operator \*(const Point &b)const

{

return x \* b.x + y \* b.y;

}

void input()

{

scanf("%d%d", &x, &y);

return ;

}

};

// 距离的平方

int dist2(Point a, Point b)

{

return (a - b) \* (a - b);

}

// 二维凸包,int

const int MAXN = 50010;

Point list[MAXN];

int Stack[MAXN], top;

bool \_cmp(Point p1, Point p2)

{

int tmp = (p1 - list[0]) ^ (p2 - list[0]);

if (tmp > 0)

{

return true;

}

else if (tmp == 0 && dist2(p1, list[0]) <= dist2(p2, list[0]))

{

return true;

}

else

{

return false;

}

}

void Graham(int n)

{

Point p0;

int k = 0;

p0 = list[0];

for (int i = 1; i < n; i++)

{

if (p0.y > list[i].y || (p0.y == list[i].y && p0.x > list[i].x))

{

p0 = list[i];

k = i;

}

}

swap(list[k], list[0]);

sort(list + 1, list + n, \_cmp);

if (n == 1)

{

top = 1;

Stack[0] = 0;

return ;

}

if (n == 2)

{

top = 2;

Stack[0] = 0;

Stack[1] = 1;

return ;

}

Stack[0] = 0;

Stack[1] = 1;

top = 2;

for (int i = 2; i < n; i++)

{

while (top > 1 && ((list[Stack[top - 1]] - list[Stack[top - 2]]) ^ (list[i] - list[Stack[top - 2]])) <= 0)

{

top--;

}

Stack[top++] = i;

}

return ;

}

int rotating\_calipers(Point p[], int n)

{

int ans = 0;

Point v;

for (int i = 0; i < n; i++)

{

int j = (i + 1) % n;

int k = (j + 1) % n;

while (j != i && k != i)

{

ans = max(ans, abs((p[j] - p[i]) ^ (p[k] - p[i])));

while (((p[i] - p[j]) ^ (p[(k + 1) % n] - p[k])) < 0)

{

k = (k + 1) % n;

}

j = (j + 1) % n;

}

}

return ans;

}

Point p[MAXN];

int main()

{

int n;

while (scanf("%d",&n) == 1)

{

if (n == -1)

{

break;

}

for (int i = 0; i < n; i++)

{

list[i].input();

}

Graham(n);

for (int i = 0; i < top; i++)

{

p[i] = list[Stack[i]];

}

printf("%.2f\n", (double)rotating\_calipers(p, top) / 2);

}

return 0;

}

##### 求解两凸包最小距离

const double eps = 1e-8;

int sgn(double x)

{

if (fabs(x) < eps)

{

return 0;

}

if (x < 0)

{

return -1;

}

else

{

return 1;

}

}

struct Point

{

double x, y;

Point(double \_x = 0, double \_y = 0)

{

x = \_x;

y = \_y;

}

Point operator - (const Point &b)const

{

return Point(x - b.x, y - b.y);

}

double operator ^ (const Point &b)const

{

return x \* b.y - y \* b.x;

}

double operator \* (const Point &b)const

{

return x \* b.x + y \* b.y;

}

void input()

{

scanf("%lf%lf", &x, &y);

}

};

struct Line

{

Point s, e;

Line(){}

Line(Point \_s, Point \_e)

{

s = \_s;

e = \_e;

}

};

// 两点间距离

double dist(Point a, Point b)

{

return sqrt((a - b) \* (a - b));

}

// 点到线段的距离,返回点到线段最近的点

Point NearestPointToLineSeg(Point P, Line L)

{

Point result;

double t = ((P - L.s) \* (L.e - L.s)) / ((L.e - L.s) \* (L.e - L.s));

if (t >=0 && t <= 1)

{

result.x = L.s.x + (L.e.x - L.s.x) \* t;

result.y = L.s.y + (L.e.y - L.s.y) \* t;

}

else

{

if (dist(P,L.s) < dist(P,L.e))

{

result = L.s;

}

else

{

result = L.e;

}

}

return result;

}

/\*

\* 求凸包,Graham算法

\* 点的编号0~n-1

\* 返回凸包结果Stack[0~top-1]为凸包的编号

\*/

const int MAXN = 10010;

Point list[MAXN];

int Stack[MAXN], top; // 相对于list[0]的极角排序

bool \_cmp(Point p1, Point p2)

{

double tmp = (p1 - list[0]) ^ (p2 - list[0]);

if (sgn(tmp) > 0)

{

return true;

}

else if (sgn(tmp) == 0 && sgn(dist(p1, list[0]) - dist(p2, list[0])) <= 0)

{

return true;

}

else

{

return false;

}

}

void Graham(int n)

{

Point p0;

int k = 0;

p0 = list[0]; // 找最下边的一个点

for (int i = 1; i < n; i++)

{

if ((p0.y > list[i].y) || (p0.y == list[i].y && p0.x > list[i].x))

{

p0 = list[i];

k = i;

}

}

swap(list[k], list[0]);

sort(list + 1, list + n, \_cmp);

if (n == 1)

{

top = 1;

Stack[0] = 0;

return ;

}

if (n == 2)

{

top = 2;

Stack[0] = 0;

Stack[1] = 1;

return ;

}

Stack[0] = 0;

Stack[1] = 1;

top = 2;

for (int i = 2; i < n; i++)

{

while (top > 1 && sgn((list[Stack[top - 1]] - list[Stack[top - 2]]) ^ (list[i] - list[Stack[top - 2]])) <= 0)

{

top--;

}

Stack[top++] = i;

}

return ;

}

// 点p0到线段p1p2的距离

double pointtoseg(Point p0, Point p1, Point p2)

{

return dist(p0, NearestPointToLineSeg(p0, Line(p1, p2)));

}

// 平行线段p0p1和p2p3的距离

double dispallseg(Point p0, Point p1, Point p2, Point p3)

{

double ans1 = min(pointtoseg(p0, p2, p3), pointtoseg(p1, p2, p3));

double ans2 = min(pointtoseg(p2, p0, p1), pointtoseg(p3, p0, p1));

return min(ans1, ans2);

}

// 得到向量a1a2和b1b2的位置关系

double Get\_angle(Point a1, Point a2, Point b1, Point b2)

{

return (a2 - a1) ^ (b1 - b2);

}

double rotating\_calipers(Point p[], int np, Point q[], int nq)

{

int sp = 0, sq = 0;

for (int i = 0; i < np; i++)

{

if (sgn(p[i].y - p[sp].y) < 0)

{

sp = i;

}

}

for (int i = 0; i < nq; i++)

{

if (sgn(q[i].y - q[sq].y) > 0)

{

sq = i;

}

}

double tmp;

double ans = dist(p[sp], q[sq]);

for (int i = 0; i < np; i++)

{

while (sgn(tmp = Get\_angle(p[sp], p[(sp + 1) % np], q[sq], q[(sq + 1) % nq])) < 0)

{

sq = (sq + 1) % nq;

}

if (sgn(tmp) == 0)

{

ans = min(ans, dispallseg(p[sp], p[(sp + 1) % np], q[sq], q[(sq + 1) % nq]));

}

else

{

ans = min(ans, pointtoseg(q[sq], p[sp], p[(sp + 1) % np]));

}

sp = (sp + 1) % np;

}

return ans;

}

double solve(Point p[], int n, Point q[], int m)

{

return min(rotating\_calipers(p, n, q, m), rotating\_calipers(q, m, p, n));

}

Point p[MAXN], q[MAXN];

int main()

{

int n, m;

while (scanf("%d%d", &n, &m) == 2)

{

if (n == 0 && m == 0)

{

break;

}

for (int i = 0; i < n; i++)

{

list[i].input();

}

Graham(n);

for (int i = 0; i < top; i++)

{

p[i] = list[i];

}

n = top;

for (int i = 0; i < m; i++)

{

list[i].input();

}

Graham(m);

for (int i = 0; i < top; i++)

{

q[i] = list[i];

}

m = top;

printf("%.4f\n", solve(p, n, q, m));

}

return 0;

}

### 半平面交

##### 半平面交模板

const double eps = 1e-8;

const double PI = acos(-1.0);

int sgn(double x)

{

if (fabs(x) < eps)

{

return 0;

}

if (x < 0)

{

return -1;

}

else

{

return 1;

}

}

struct Point

{

double x, y;

Point(){}

Point(double \_x, double \_y)

{

x = \_x;

y = \_y;

}

Point operator - (const Point &b)const

{

return Point(x - b.x, y - b.y);

}

double operator ^ (const Point &b)const

{

return x \* b.y - y \* b.x;

}

double operator \* (const Point &b)const

{

return x \* b.x + y \* b.y;

}

};

struct Line

{

Point s, e;

double k;

Line() {}

Line(Point \_s, Point \_e)

{

s = \_s;

e = \_e;

k = atan2(e.y - s.y, e.x - s.x);

}

Point operator & (const Line &b)const

{

Point res = s;

double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));

res.x += (e.x - s.x) \* t;

res.y += (e.y - s.y) \* t;

return res;

}

};

// 半平面交,直线的左边代表有效区域

bool HPIcmp(Line a, Line b)

{

if (fabs(a.k - b.k) > eps)

{

return a.k < b.k;

}

return ((a.s - b.s) ^ (b.e - b.s)) < 0;

}

Line Q[110];

void HPI(Line line[], int n, Point res[], int &resn)

{

int tot = n;

sort(line, line + n, HPIcmp);

tot = 1;

for (int i = 1; i < n; i++)

{

if (fabs(line[i].k - line[i-1].k) > eps)

{

line[tot++] = line[i];

}

}

int head = 0, tail = 1;

Q[0] = line[0];

Q[1] = line[1];

resn = 0;

for (int i = 2; i < tot; i++)

{

if (fabs((Q[tail].e - Q[tail].s) ^ (Q[tail - 1].e - Q[tail - 1].s)) < eps || fabs((Q[head].e - Q[head].s) ^ (Q[head + 1].e - Q[head + 1].s)) < eps)

{

return;

}

while (head < tail && (((Q[tail] & Q[tail - 1]) - line[i].s) ^ (line[i].e - line[i].s)) > eps)

{

tail--;

}

while (head < tail && (((Q[head] & Q[head + 1]) - line[i].s) ^ (line[i].e - line[i].s)) > eps)

{

head++;

}

Q[++tail] = line[i];

}

while (head < tail && (((Q[tail] & Q[tail - 1]) - Q[head].s) ^ (Q[head].e - Q[head].s)) > eps)

{

tail--;

}

while (head < tail && (((Q[head]&Q[head-1]) - Q[tail].s) ^ (Q[tail].e - Q[tail].e)) > eps)

{

head++;

}

if (tail <= head + 1)

{

return ;

}

for (int i = head; i < tail; i++)

{

res[resn++] = Q[i] & Q[i + 1];

}

if (head < tail - 1)

{

res[resn++] = Q[head]&Q[tail];

}

return ;

}

##### 普通半平面交写法

const double eps = 1e-18;

int sgn(double x)

{

if (fabs(x) < eps)

{

return 0;

}

if (x < 0)

{

return -1;

}

else

{

return 1;

}

}

struct Point

{

double x, y;

Point() {}

Point(double \_x, double \_y)

{

x = \_x;

y = \_y;

}

Point operator - (const Point &b)const

{

return Point(x - b.x, y - b.y);

}

double operator ^ (const Point &b)const

{

return x \* b.y - y \* b.x;

}

double operator \* (const Point &b)const

{

return x \* b.x + y \* b.y;

}

};

// 计算多边形面积

double CalcArea(Point p[], int n)

{

double res = 0;

for (int i = 0; i < n; i++)

{

res += (p[i] ^ p[(i + 1) % n]);

}

return fabs(res / 2);

}

// 通过两点,确定直线方程

void Get\_equation(Point p1, Point p2, double &a, double &b, double &c)

{

a = p2.y - p1.y;

b = p1.x - p2.x;

c = p2.x \* p1.y - p1.x \* p2.y;

return ;

}

// 求交点

Point Intersection(Point p1, Point p2, double a, double b, double c)

{

double u = fabs(a \* p1.x + b \* p1.y + c);

double v = fabs(a \* p2.x + b \* p2.y + c);

Point t;

t.x = (p1.x \* v + p2.x \* u) / (u + v);

t.y = (p1.y \* v + p2.y \* u) / (u + v);

return t;

}

Point tp[110];

void Cut(double a, double b, double c, Point p[], int &cnt)

{

int tmp = 0;

for (int i = 1; i <= cnt; i++)

{

// 当前点在左侧,逆时针的点

if (a \* p[i].x + b \* p[i].y + c < eps)

{

tp[++tmp] = p[i];

}

else

{

if (a \* p[i - 1].x + b \* p[i - 1].y + c < -eps)

{tp[++tmp] = Intersection(p[i - 1], p[i], a, b, c);

}

if (a \* p[i + 1].x + b \* p[i + 1].y + c < -eps)

{

tp[++tmp] = Intersection(p[i], p[i + 1], a, b, c);

}

}

}

for (int i = 1; i <= tmp; i++)

{

p[i] = tp[i];

}

p[0] = p[tmp];

p[tmp + 1] = p[1];

cnt = tmp;

return ;

}

double V[110], U[110], W[110];

int n;

const double INF = 100000000000.0;

Point p[110];

bool solve(int id)

{

p[1] = Point(0, 0);

p[2] = Point(INF, 0);

p[3] = Point(INF, INF);

p[4] = Point(0, INF);

p[0] = p[4];

p[5] = p[1];

int cnt = 4;

for (int i = 0; i < n; i++)

{

if (i != id)

{

double a = (V[i] - V[id]) / (V[i] \* V[id]);

double b = (U[i] - U[id]) / (U[i] \* U[id]);

double c = (W[i] - W[id]) / (W[i] \* W[id]);

if (sgn(a) == 0 && sgn(b) == 0)

{

if (sgn(c) >= 0)

{

return false;

}

else

{

continue;

}

}

Cut(a, b, c, p, cnt);

}

}

if (sgn(CalcArea(p, cnt)) == 0)

{

return false;

}

else

{

return true;

}

}

int main()

{

while (scanf("%d", &n) == 1)

{

for (int i = 0; i < n; i++)

{

scanf("%lf%lf%lf", &V[i], &U[i], &W[i]);

}

for (int i = 0; i < n; i++)

{

if (solve(i))

{

printf("Yes\n");

}

else

{

printf("No\n");

}

}

}

return 0;

}

### 计算几何相关公式

##### Pick公式

顶点坐标均是整点的简单多边形:   
面积 = 内部格点数目 + 边上格点数目 / 2 – 1

S = n + s / 2 - 1   
(其中n表示多边形内部的点数,s表示多边形边界上的点数,S表示多边形的面积)

##### 已知圆锥表面积S求最大体积V

V = S \* sqrt(S / (72 \* Pi))

### Liuctic计算几何库

/\*

\* Liuctic 的计算几何库

\* p-Lpoint ln, l - Lline ls - Llineseglr - Lrad

\* 平面上两点之间的距离 p2pdis

\* (P1-P0)\*(P2-P0)的叉积 xmulti

\* 确定两条线段是否相交 lsinterls

\* 判断点p是否在线段l上 ponls

\* 判断两个点是否相等 Euqal\_Point

\* 线段非端点相交 lsinterls\_A

\* 判断点q是否在多边形Polygon内 pinplg

\* 多边形的面积 area\_of\_polygon

\* 解二次方程 Ax^2+Bx+C=0 equa

\* 计算直线的一般式 Ax+By+C=0 format

\* 点到直线距离 p2lndis

\* 直线与圆的交点,已知直线与圆相交 lncrossc

\* 点是否在射线的正向 samedir

\* 射线与圆的第一个交点 lrcrossc

\* 求点p1关于直线ln的对称点p2 mirror

\* 两直线夹角(弧度) angle\_LL

\* 求两圆相交的面积 Area\_of\_overlap

\* 求两矩形相交的面积 Area\_of\_overlap\_rec

\*/

#define infinity 1e20

#define EP 1e-10

const int MAXV = 300;

const double PI = 2.0 \* asin(1.0); // 高精度PI

struct Lpoint

{

double x, y;

}; // 点

struct Llineseg

{

Lpoint a, b;

}; // 线段

struct Ldir

{

double dx, dy;

}; // 方向向量

struct Lline

{

Lpoint p;

Ldir dir;

}; // 直线

struct Lrad

{

Lpoint Sp;

Ldir dir;

}; // 射线

struct Lround

{

Lpoint co;

double r;

}; // 圆

##### 求平面两点之间的距离

double p2pdis(Lpoint p1, Lpoint p2)

{

return (sqrt((p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y)));

}

##### (P1-P0)\*(P2-P0)的叉积

/\*

\* 若结果为正,则<P0,P1>在<P0,P2>的顺时针方向;

\* 若为0则<P0,P1><P0,P2>共线;

\* 若为负则<P0,P1>在<P0,P2>的在逆时针方向;

\* 可以根据这个函数确定两条线段在交点处的转向,比如确定p0p1和p1p2在p1处是左转还是右转,只要求(p2-p0)\*(p1-p0),

\* 若<0则左转,>0则右转,=0则共线

\*/

double xmulti(Lpoint p1, Lpoint p2, Lpoint p0)

{

return ((p1.x - p0.x) \* (p2.y - p0.y) - (p2.x - p0.x) \* (p1.y - p0.y));

}

##### 确定两条线段是否相交

double mx(double t1, double t2)

{

if (t1 > t2)

{

return t1;

}

return t2;

}

double mn(double t1, double t2)

{

if (t1 < t2)

{

return t1;

}

return t2;

}

int lsinterls(Llineseg u, Llineseg v)

{

return ((mx(u.a.x, u.b.x) >= mn(v.a.x, v.b.x)) && (mx(v.a.x, v.b.x) >= mn(u.a.x, u.b.x)) && (mx(u.a.y, u.b.y) >= mn(v.a.y, v.b.y)) && (mx(v.a.y, v.b.y) >= mn(u.a.y, u.b.y)) && (xmulti(v.a, u.b, u.a) \* xmulti(u.b, v.b, u.a) >= 0) && (xmulti(u.a, v.b, v.a) \* xmulti(v.b, u.b, v.a) >= 0));

}

##### 判断点p是否在线段l上

int ponls(Llineseg l, Lpoint p)

{

return ((xmulti(l.b, p, l.a) == 0) && (((p.x - l.a.x) \* (p.x - l.b.x) < 0) || ((p.y - l.a.y) \* (p.y - l.b.y) < 0)));

}

##### 判断两个点是否相等

int Euqal\_Point(Lpoint p1, Lpoint p2)

{

return ((fabs(p1.x - p2.x) < EP) && (fabs(p1.y - p2.y) < EP));

}

##### 线段相交判断函数

/\*

\* 当且仅当u,v相交并且交点不是u,v的端点时函数为true;

\*/

int lsinterls\_A(Llineseg u, Llineseg v)

{

return ((lsinterls(u, v)) && (!Euqal\_Point(u.a, v.a)) && (!Euqal\_Point(u.a, v.b)) && (!Euqal\_Point(u.b, v.a)) && (!Euqal\_Point(u.b, v.b)));

}

##### 判断点q是否在多边形内

/\*

\* 其中多边形是任意的凸或凹多边形,

\* Polygon中存放多边形的逆时针顶点序列

\*/

int pinplg(int vcount, Lpoint Polygon[], Lpoint q)

{

int c = 0, i, n;

Llineseg l1, l2;

l1.a = q;

l1.b = q;

l1.b.x = infinity;

n = vcount;

for (i = 0; i < vcount; i++)

{

l2.a = Polygon[i];

l2.b = Polygon[(i + 1) % n];

if ((lsinterls\_A(l1, l2)) || ((ponls(l1, Polygon[(i + 1) % n])) && (((!ponls(l1, Polygon[(i + 2) % n])) && (xmulti(Polygon[i], Polygon[(i + 1) % n], l1.a) \* xmulti(Polygon[(i + 1) % n], Polygon[(i + 2) % n], l1.a) > 0)) || ((ponls(l1, Polygon[(i + 2) % n])) && (xmulti(Polygon[i], Polygon[(i + 2) % n], l1.a) \* xmulti(Polygon[(i + 2) % n], Polygon[(i + 3) % n], l1.a) > 0)))))

{

c++;

}

}

return (c % 2 != 0);

}

##### 多边形的面积

/\*

\* 要求按照逆时针方向输入多边形顶点

\* 可以是凸多边形或凹多边形

\*/

double area\_of\_polygon(int vcount, Lpoint plg[])

{

int i;

double s;

if (vcount < 3)

{

return 0;

}

s = plg[0].y \* (plg[vcount - 1].x - plg[1].x);

for (i = 1; i < vcount; i++)

{

s += plg[i].y \* (plg[(i - 1)].x - plg[(i + 1) % vcount].x);

}

return s / 2;

}

##### 解二次方程 Ax^2+Bx+C=0

/\*

\* 返回-1表示无解 返回1 表示有解

\*/

int equa(double A, double B, double C, double &x1, double &x2)

{

double f = B \* B - 4 \* A \* C;

if (f < 0)

{

return -1;

}

x1 = (-B + sqrt(f)) / (2 \* A);

x2 = (-B - sqrt(f)) / (2 \* A);

return 1;

}

##### 计算直线的一般式 Ax+By+C=0

void format(Lline ln, double &A, double &B, double &C)

{

A = ln.dir.dy;

B = -ln.dir.dx;

C = ln.p.y \* ln.dir.dx - ln.p.x \* ln.dir.dy;

return ;

}

##### 点到直线的距离

double p2lndis(Lpoint a, Lline ln)

{

double A, B, C;

format(ln, A, B, C);

return (fabs(A \* a.x + B \* a.y + C) / sqrt(A \* A + B \* B));

}

##### 直线与圆的交点,已知直线与圆相交

int lncrossc(Lline ln, Lround Y, Lpoint &p1, Lpoint &p2)

{

double A, B, C, t1, t2;

int zz = -1;

format(ln, A, B, C);

if (fabs(B) < 1e-8)

{

p1.x = p2.x = -1.0 \* C / A;

zz = equa(1.0, -2.0 \* Y.co.y, Y.co.y \* Y.co.y + (p1.x - Y.co.x) \* (p1.x - Y.co.x) - Y.r \* Y.r, t1, t2);

p1.y = t1;

p2.y = t2;

}

else if (fabs(A) < 1e-8)

{

p1.y = p2.y = -1.0 \* C / B;

zz = equa(1.0, -2.0 \* Y.co.x, Y.co.x \* Y.co.x + (p1.y - Y.co.y) \* (p1.y - Y.co.y) - Y.r \* Y.r, t1, t2);

p1.x = t1;

p2.x = t2;

}

else

{

zz = equa(A \* A + B \* B, 2.0 \* A \* C + 2.0 \* A \* B \* Y.co.y - 2.0 \* B \* B \* Y.co.x, B \* B \* Y.co.x \* Y.co.x + C \* C + 2\* B \* C \* Y.co.y + B \* B \* Y.co.y \* Y.co.y - B \* B \* Y.r \* Y.r, t1, t2);

p1.x = t1, p1.y = -1 \* (A / B \* t1 + C / B);

p2.x = t2, p2.y = -1 \* (A / B \* t2 + C / B);

}

return 0;

}

##### 点是否在射线的正向

bool samedir(Lrad ln, Lpoint P)

{

double ddx, ddy;

ddx = P.x - ln.Sp.x;

ddy = P.y - ln.Sp.y;

if ((ddx \* ln.dir.dx > 0 || fabs(ddx \* ln.dir.dx) < 1e-7) && (ddy \* ln.dir.dy > 0 || (fabs(ddy \* ln.dir.dy) < 1e-7)))

{

return true;

}

else

{

return false;

}

}

##### 射线与圆的第一个交点

/\*

\* 已经确定射线所在直线与圆相交返回-1表示不存正向交点,否则返回1

\*/

int lrcrossc(Lrad ln, Lround Y, Lpoint &P)

{

Lline ln2;

Lpoint p1, p2;

int res = -1;

double dis = 1e20;

ln2.p = ln.Sp, ln2.dir = ln.dir;

lncrossc(ln2, Y, p1, p2);

if (samedir(ln, p1))

{

res = 1;

if (p2pdis(p1, ln.Sp) < dis)

{

dis = p2pdis(p1, ln.Sp);

}

P = p1;

}

if (samedir(ln, p2))

{

res = 1;

if (p2pdis(p2, ln.Sp) < dis)

{

dis = p2pdis(p2, ln.Sp);

P = p2;

}

}

return res;

}

##### 求点p1关于直线ln的对称点p2

Lpoint mirror(Lpoint P, Lline ln)

{

Lpoint Q;

double A, B, C;

format(ln, A, B, C);

Q.x = ((B \* B - A \* A) \* P.x - 2 \* A \* B \* P.y - 2 \* A \* C) / (A \* A + B \* B);

Q.y = ((A \* A - B \* B) \* P.y - 2 \* A \* B \* P.x - 2 \* B \* C) / (A \* A + B \* B);

return Q;

}

##### 两直线夹角(弧度)

double angle\_LL(Lline line1, Lline line2)

{

double A1, B1, C1;

format(line1, A1, B1, C1);

double A2, B2, C2;

format(line2, A2, B2, C2);

if (A1 \* A2 + B1 \* B2 == 0)

{

return PI / 2.0; // 垂直

}

else

{

double t = fabs((A1 \* B2 - A2 \* B1) / (A1 \* A2 + B1 \* B2));

return atan(t);

}

}

##### 求两圆相交的面积

double Area\_of\_overlap(Point c1, double r1, Point c2, double r2)

{

double d = dist(c1, c2);

if (r1 + r2 < d + eps)

{

return 0;

}

if (d < fabs(r1 - r2) + eps)

{

double r = min(r1, r2);

return PI \* r \* r;

}

double x = (d \* d + r1 \* r1 - r2 \* r2) / (2 \* d);

double t1 = acos(x / r1);

double t2 = acos((d - x) / r2);

return r1 \* r1 \* t1 + r2 \* r2 \* t2 - d \* r1 \* sin(t1);

}

##### 求两矩形相交的面积

/\*

\* x[]、y[]存储矩阵对角线顶点（只需要任意一条）

\*/

double Area\_of\_overlap\_rec(double x[], double y[])

{

// 将两个矩形全部统一为主对角线

sort(x, x + 2);

sort(x + 2, x + 4);

sort(y , y + 2);

sort(y + 2, y + 4);

if (x[1] <= x[2] || x[0] >= x[3] || y[0] >= y[3] || y[1] <= y[2]) // 相离

{

return 0.0;

}

else

{

sort(x, x + 4);

sort(y, y + 4);

return (x[2] - x[1]) \* (y[2] - y[1]);

}

}

### 向量基本用法

struct node {

double x; // 横坐标

double y; // 纵坐标

};

typedef node Vector;

Vector operator + (Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }

Vector operator - (Point A, Point B) { return Vector(A.x - B.y, A.y - B.y); }

Vector operator \* (Vector A, double p) { return Vector(A.x\*p, A.y\*p); }

Vector operator / (Vector A, double p) { return Vector(A.x / p, A.y\*p); }

double Dot(Vector A, Vector B) { return A.x\*B.x + A.y\*B.y; } // 向量点乘

double Length(Vector A) { return sqrt(Dot(A, A)); } // 向量模长

double Angle(Vector A, Vector B) { return acos(Dot(A, B) / Length(A) / Length(B)); } // 向量之间夹角

double Cross(Vector A, Vector B) { // 叉积计算 公式

return A.x\*B.y - A.y\*B.x;

}

Vector Rotate(Vector A, double rad) // 向量旋转 公式 {

return Vector(A.x\*cos(rad) - A.y\*sin(rad), A.x\*sin(rad) + A.y\*cos(rad));

}

Point getLineIntersection(Point P, Vector v, Point Q, Vector w) { // 两直线交点t1 t2计算公式

Vector u = P - Q;

double t = Cross(w, u) / Cross(v, w); // 求得是横坐标

return P + v\*t; // 返回一个点

}

### 求多边形面积

node G[maxn];

int n;

double Cross(node a, node b) { // 叉积计算

return a.x\*b.y - a.y\*b.x;

}

int main()

{

while (scanf("%d", &n) != EOF && n) {

for (int i = 0; i < n; i++)

scanf("%lf %lf", &G[i].x, &G[i].y);

double sum = 0;

G[n].x = G[0].x;

G[n].y = G[0].y;

for (int i = 0; i < n; i++) {

sum += Cross(G[i], G[i + 1]);

}

// 或者

//for (int i = 0; i < n; i++) {

//sum += fun(G[i], G[（i + 1）% n]);

//}

sum = sum / 2.0;

printf("%.1f\n", sum);

}

system("pause");

return 0;

}

### 判断线段相交

node P[35][105];

double Cross\_Prouct(node A,node B,node C) { // 计算BA叉乘CA

return (B.x-A.x)\*(C.y-A.y)-(B.y-A.y)\*(C.x-A.x);

}

bool Intersect(node A,node B,node C,node D) { // 通过叉乘判断线段是否相交；

if(min(A.x,B.x)<=max(C.x,D.x)&& // 快速排斥实验；

min(C.x,D.x)<=max(A.x,B.x)&&

min(A.y,B.y)<=max(C.y,D.y)&&

min(C.y,D.y)<=max(A.y,B.y)&&

Cross\_Prouct(A,B,C)\*Cross\_Prouct(A,B,D)<0&& // 跨立实验；

Cross\_Prouct(C,D,A)\*Cross\_Prouct(C,D,B)<0) // 叉乘异号表示在两侧；

return true;

else return false;

}

### 求三角形外心

Point circumcenter(const Point &a, const Point &b, const Point &c) { //返回三角形的外心

Point ret;

double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1\*a1 + b1\*b1) / 2;

double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2\*a2 + b2\*b2) / 2;

double d = a1\*b2 - a2\*b1;

ret.x = a.x + (c1\*b2 - c2\*b1) / d;

ret.y = a.y + (a1\*c2 - a2\*c1) / d;

return ret;

}

### 极角排序

double cross(point p1, point p2, point q1, point q2) { // 叉积计算

return (q2.y - q1.y)\*(p2.x - p1.x) - (q2.x - q1.x)\*(p2.y - p1.y);

}

bool cmp(point a, point b) {

point o;

o.x = o.y = 0;

return cross(o, b, o, a) < 0; // 叉积判断

}

sort(convex + 1, convex + cnt, cmp); // 按角排序, 从小到大

# Other 其它

### 数据类型的取值范围



### 输入输出外挂

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

typedef unsigned long long ull;

typedef long double ld;

#define mp make\_pair

#define PI pair<int,int>

#define poly vector<ll>

#define For(i,l,r) for(int i=(int)(l);i<=(int)(r);i++)

#define Rep(i,r,l) for(int i=(int)(r);i>=(int)(l);i--)

#define pb push\_back

#define fi first

#define se second

inline char gc(){

static char buf[100000],\*p1=buf,\*p2=buf;

return p1==p2&&(p2=(p1=buf)+fread(buf,1,100000,stdin),p1==p2)?EOF:\*p1++;

}

#define gc getchar

inline ll read(){

ll x = 0; char ch = gc(); bool positive = 1;

for (; !isdigit(ch); ch = gc()) if (ch == '-') positive = 0;

for (; isdigit(ch); ch = gc()) x = x \* 10 + ch - '0';

return positive ? x : -x;

}

inline void write(ll a){

if(a<0){

a=-a; putchar('-');

}

if(a>=10)write(a/10);

putchar('0'+a%10);

}

inline void writeln(ll a){write(a); puts("");}

inline void wri(ll a){write(a); putchar(' ');}

inline ull rnd(){

return ((ull)rand()<<30^rand())<<4|rand()%4;

}

const int mod=998244353;

int ans=1;

int main(){

int h=read(),w=read();

For(i,1,h+w)ans=ans\*2%mod;

cout<<ans<<endl;

}

/\*

dp[j][i]=max(dp[j^1][k]+s[j][i]-s[j][k])

\*/

### 关于爆栈问题——手动加栈

##### 解决爆栈，手动加栈

当我们用C++时，可以用上边的这句代码进行手动加栈。（#pragma comment(linker, “/STACK:1024000000,1024000000”)）

而C时，我们则要说另外一句代码：#pragma GCC optimize (“O2”)

我想，对于这个02你应该会有一丝的警觉，02表示一种状态，那么是不是还有其他的状态呢？

对，这个真有。GCC的#pragma优化主要分为四种 -O0 -O1 -O2 -O3

-O0 表示无优化状态

-O1 表示对代码进行了优化

-O2 表示减小目标文件大小

-O3 表示减小代码段及栈空间的大小

对部分代码可以去除优化：

#pragma GCC push\_options

#pragma GCC optimize (“O0”)

或者也可以增加优化

#pragma GCC pop\_options

# 重要公式与定理

### 31天的月份

int mon[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};

31天的月份 1 3 5 7 8 10 12

### 组合公式C(n,m)

int C(int n,int m)

{

int res=1;

if(n<m)

return 0;

else

{

if(m>n/2)

m=n-m;

for(int i=1;i<=m;i++)

res=res\*(n-m+i)/i;

return res;

}

}

### 全排列

void Pern(int list[], int k, int n) { // k表示前k个数不动仅移动后面n-k位数

if (k == n - 1) {

for (int i = 0; i < n; i++) {

printf("%d", list[i]);

}

printf("\n");

}else {

for (int i = k; i < n; i++) { // 输出的是满足移动条件所有全排列

swap(list[k], list[i]);

Pern(list, k + 1, n);

swap(list[k], list[i]);

}

}

}

### 错排公式

void f()

{

a[1]=0;

a[2]=1;

for(int i=3;i<maxn;i++)

{

a[i]=(i-1)\*(a[i-1]+a[i-2]);

}

}

### 递推公式-阿牛的EOF牛肉串

准备在上面刻下一个长度为n的只由"E" “O” "F"三种字符组成的字符串（可以只有其中一种或两种字符，但绝对不能有其他字符）,阿牛同时禁止在串中出现O相邻的情况，他认为，"OO"看起来就像发怒的眼睛，效果不好。

第一种情况：

第 n 个格子放 O 那么由题目条件限制 第 n - 1 个格子只能放E 或者 F

n - 2 与约束无关 那么此时的方案数等于 2 \* f ( n - 2 )

第二种情况：

第 n 个格子不放 O 那么第 n个格子就可以取 E 或者 F 了

n - 1 与约束无关 那么此时的方案书等于 2 \* f ( n - 1 )

总的方案数 f (n ) = 2 \* f ( n - 1 ) + 2 \* f ( n - 2 )

### 递推公式-红粉绿涂色

有排成一行的ｎ个方格，用红(Red)、粉(Pink)、绿(Green)三色涂每个格子，每格涂一色，要求任何相邻的方格不能同色，且首尾两格也不同色．求全部的满足要求的涂法.

第一种情况：

对于前 n - 1个方格，如果它是合理分布的 即相邻的方格不能同色，且首尾两格也不同色

那么第 n 种涂色 和第 n - 1 方案数是一样的 因为第一个和 n - 1颜色不相同

已经确定了 那么第 n 个颜色就已经确定了 那么方案数就是 f (n - 1 )

第二种情况：

对于前 n - 1个方格，如果它不是合理分布的 我们不需要考虑中间怎么安排 不合理就是首尾两格颜色是相同的 显然，这种方案数是不存在的 此时的方案数应该和 n - 2 项是相同的

那么对于第 n 个格子我们就有两种颜色选择了 那么方案书就是2 \* f ( n - 2 )

总的方案数 f ( n ) = f (n - 1 ) + 2 \* f ( n - 2 )

### 汉诺塔③-三根杆

我们设f(n)为把n个盘从左(1)移到3所需要的步数，当然也等于从3移到1的步数。

可知，要想把第n个盘从左(1)移到3，需要想把前n-1个从左(1)移动右(3)，再从右(3)移到左(1)，最后再从左(1)移到右(3)。

而第n个盘要从左(1)到中(2)再右(3)经历2步。

所以，f(n)=3\*f(n-1)+2;

经数学计算最终可得到f(n)=3^n-1;

### 迷宫，dp，卡特兰数

从起点(0，0)走到终点(n,n)的最短路径数是C(2n,n),现在小兔又想如果不穿越对角线(但可接触对角线上的格点)，这样的路径数有多少

#include<bits/stdc++.h>

using namespace std;

const int maxn=40;

long long dp[maxn][maxn];

int main()

{

memset(dp,0,sizeof(dp));

for(int i=1;i<maxn;i++)

{

dp[i][0]=1;

for(int j=1;j<i;j++)

{

dp[i][j]=dp[i-1][j]+dp[i][j-1];

}

dp[i][i]=dp[i][i-1];

}

int k=0;

int n;

while(cin>>n&&n!=-1)

{

cout<<++k<<" "<<n<<" "<<2\*dp[n][n]<<endl;

}

return 0;

}

### 卡特兰数 h(n)=C(2n,n)/(n+1) (n=0,1,2,...)  h(n)=c(2n,n)-c(2n,n-1)(n=0,1,2,...)  C(m+n,n)−C(m+n,n−1)

### 欧几里得求最小公倍数

#include<bits/stdc++.h>

using namespace std;

const int maxn=1000+5;

long long a[maxn];

long long gcd(long long a,long long b) //求最大公约数

{

if(a%b==0) return b;

else return gcd(b,a%b);

}

long long lcm(long long a,long long b) //求最小公倍数

{

return (a\*b)/gcd(a,b);

}

int main()

{

long long n;

while(cin>>n&&n){

for(int i=0;i<n;i++)

cin>>a[i];

long long res=a[0];

for(int i=0;i<n;i++)

res=lcm(res,a[i]);

cout<<res<<endl;

}

return 0;

}

# 模拟题

### 叠筐

5 @ W

@@@  
@WWW@  
@W@W@  
@WWW@  
 @@@

#include<bits/stdc++.h>

using namespace std;

const int maxn=80+5;

char g[maxn][maxn];

int main()

{

int n;

char cen;

char out;

char cc;

int space=0;

while(cin>>n>>cen>>out)

{

if(space)

{

cout<<endl;

}

int cnt=0;

int k=(n+1)/2;

for(int i=k;i>=1;i--)

{

if(cnt%2==0)

cc=cen;

else

cc=out;

for(int j=i;j<=n-i+1;j++)

{

g[i][j]=g[n-i+1][j]=cc;

g[j][i]=g[j][n-i+1]=cc;

}

++cnt;

}

if(n!=1)

{

g[1][1]=g[1][n]=g[n][1]=g[n][n]=' ';

}

for(int i=1;i<=n;i++)

{

for(int j=1;j<=n;j++)

{

cout<<g[i][j];

}

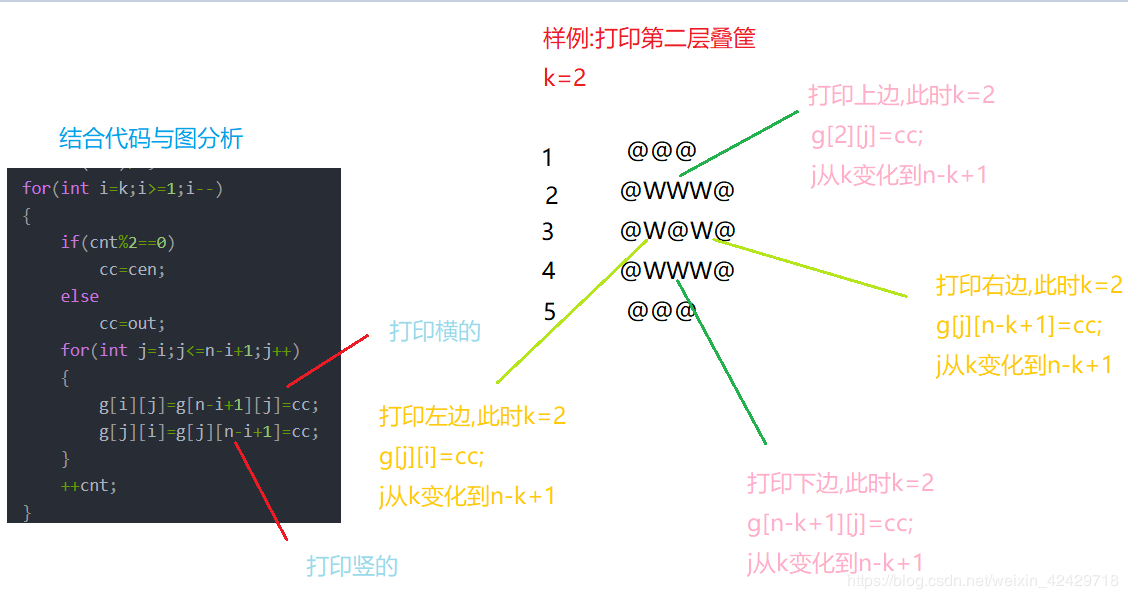
cout<<endl;

}

space++;

}

return 0;

}

# 二分图

### 染色法

|交叉染色法判断二分图|

int bipartite(int s) {

int u, v;

queue<int>Q;

color[s] = 1;

Q.push(s);

while (!Q.empty()) {

u = Q.front();

Q.pop();

for (int i = 0; i < G[u].size(); i++) {

v = G[u][i];

if (color[v] == 0) {

color[v] = -color[u];

Q.push(v);

}

else if (color[v] == color[u])

return 0;

}

}

return 1;

}

### 二分图最大权匹配，KM算法

输入数据包含多组测试用例，每组数据的第一行输入n,表示房子的数量(也是老百姓家的数量)，接下来有n行,每行n个数表示第i个村名对第j间房出的价格(n<=300)。

#include <iostream>

#include <cstring>

#include <cstdio>

using namespace std;

const int MAXN = 305;

const int INF = 0x3f3f3f3f;

int love[MAXN][MAXN]; // 记录每个妹子和每个男生的好感度

int ex\_girl[MAXN]; // 每个妹子的期望值

int ex\_boy[MAXN]; // 每个男生的期望值

bool vis\_girl[MAXN]; // 记录每一轮匹配匹配过的女生

bool vis\_boy[MAXN]; // 记录每一轮匹配匹配过的男生

int match[MAXN]; // 记录每个男生匹配到的妹子 如果没有则为-1

int slack[MAXN]; // 记录每个汉子如果能被妹子倾心最少还需要多少期望值

int N;

bool dfs(int girl)

{

vis\_girl[girl] = true;

for (int boy = 0; boy < N; ++boy) {

if (vis\_boy[boy]) continue; // 每一轮匹配 每个男生只尝试一次

int gap = ex\_girl[girl] + ex\_boy[boy] - love[girl][boy];

if (gap == 0) { // 如果符合要求

vis\_boy[boy] = true;

if (match[boy] == -1 || dfs( match[boy] )) { // 找到一个没有匹配的男生 或者该男生的妹子可以找到其他人

match[boy] = girl;

return true;

}

} else {

slack[boy] = min(slack[boy], gap); // slack 可以理解为该男生要得到女生的倾心 还需多少期望值 取最小值 备胎的样子【捂脸

}

}

return false;

}

int KM()

{

memset(match, -1, sizeof match); // 初始每个男生都没有匹配的女生

memset(ex\_boy, 0, sizeof ex\_boy); // 初始每个男生的期望值为0

// 每个女生的初始期望值是与她相连的男生最大的好感度

for (int i = 0; i < N; ++i) {

ex\_girl[i] = love[i][0];

for (int j = 1; j < N; ++j) {

ex\_girl[i] = max(ex\_girl[i], love[i][j]);

}

}

// 尝试为每一个女生解决归宿问题

for (int i = 0; i < N; ++i) {

fill(slack, slack + N, INF); // 因为要取最小值 初始化为无穷大

while (1) {

// 为每个女生解决归宿问题的方法是 ：如果找不到就降低期望值，直到找到为止

// 记录每轮匹配中男生女生是否被尝试匹配过

memset(vis\_girl, false, sizeof vis\_girl);

memset(vis\_boy, false, sizeof vis\_boy);

if (dfs(i)) break; // 找到归宿 退出

// 如果不能找到 就降低期望值

// 最小可降低的期望值

int d = INF;

for (int j = 0; j < N; ++j)

if (!vis\_boy[j]) d = min(d, slack[j]);

for (int j = 0; j < N; ++j) {

// 所有访问过的女生降低期望值

if (vis\_girl[j]) ex\_girl[j] -= d;

// 所有访问过的男生增加期望值

if (vis\_boy[j]) ex\_boy[j] += d;

// 没有访问过的boy 因为girl们的期望值降低，距离得到女生倾心又进了一步！

else slack[j] -= d;

}

}

}

// 匹配完成 求出所有配对的好感度的和

int res = 0;

for (int i = 0; i < N; ++i)

res += love[ match[i] ][i];

return res;

}

int main()

{

while (~scanf("%d", &N)) {

for (int i = 0; i < N; ++i)

for (int j = 0; j < N; ++j)

scanf("%d", &love[i][j]);

printf("%d\n", KM());

}

return 0;

}

### 二分图**最小权值匹配，KM算法**

**有n个盒子摆成的环，每个盒子里有一定数量的巧克力，巧克力总数小于n，问最少移几步(每步等于把一块巧克力从第i格移动到了i+1或i-1格)，使得每个盒子里巧克力数小于等于1，从第i个盒子移到第j个盒子的最小步数为min( n-abs(i-j), abs(i-j) )。**

如果一个盒子里有5块巧克力,其实我们需要做的就是把这盒子里的4块巧克力分别移到4个空盒子里去.

建立二分图: 左边点集是每一块需要被移动的巧克力, 右边点集是每一个空的盒子. 对于左边第i块巧克力来说,它被移动到右边第j个空盒子的最小距离为X,那么就在左i点与右j点之间加一条X权值的边.

最终我们求得就是该二分图的最小权值匹配.(权值取负数就是最优权值匹配了:)

Sample Input

In 10 1 3 3 0 0 2 0 0 0 0 out 0

#include<iostream>

#include<cstring>

#include<cmath>

#define mst(a,b) memset(a,b,sizeof(a))

using namespace std;

const int maxn=500+8;

const int INF=0x3f3f3f3f;

int a[maxn],x[maxn],y[maxn];

int love[maxn][maxn]; // 记录每个妹子和每个男生的好感度

int ex\_girl[maxn]; // 每个妹子的期望值

int ex\_boy[maxn]; // 每个男生的期望值

bool vis\_girl[maxn]; // 记录每一轮匹配匹配过的女生

bool vis\_boy[maxn]; // 记录每一轮匹配匹配过的男生

int match[maxn]; // 记录每个男生匹配到的妹子 如果没有则为-1

int slack[maxn]; // 记录每个汉子如果能被妹子倾心最少还需要多少期望值

int n,nx,ny;

bool dfs(int girl)

{

vis\_girl[girl]=true;

for(int boy=1;boy<=ny;++boy){

if(vis\_boy[boy]) continue;

int gap=ex\_boy[boy]+ex\_girl[girl]-love[girl][boy];

if(gap==0)

{

vis\_boy[boy]=true;

if(match[boy]==-1||dfs(match[boy]))

{

match[boy]=girl;

return true;

}

}

else

{

slack[boy]=min(slack[boy],gap);

}

}

return false;

}

int KM()

{

mst(match,-1);

mst(ex\_boy,0);

for(int i=1;i<=nx;i++)

{

ex\_girl[i]=love[i][0];

for(int j=2;j<=ny;j++)

{

ex\_girl[i]=max(ex\_girl[i],love[i][j]);

}

}

for(int i=1;i<=nx;i++)

{

fill(slack+1,slack+n+1,INF);

while(true)

{

mst(vis\_boy,false);

mst(vis\_girl,false);

if(dfs(i)) break;

int d=INF;

for(int j=1;j<=ny;j++)

if(!vis\_boy[j]) d=min(d,slack[j]);

for(int j=1;j<=nx;j++)

if(vis\_girl[j]) ex\_girl[j]-=d;

for(int j=1;j<=ny;j++){

if(vis\_boy[j]) ex\_boy[j]+=d;

else

slack[j]-=d;

}

}

}

int res=0;

for(int i=1;i<=ny;i++) //注意这里是ny 不是nx

//match数组定义:记录每个男生匹配到的妹子 如果没有则为-1

if(match[i]!=-1)

res+=love[match[i]][i];

return res;

}

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

while(cin>>n)

{

mst(love,0); //初始化我们二分图的权值边

nx=0;ny=0; //nx用来记录每个盒子多的巧克力数

//ny用来记录空盒子数

for(int i=1;i<=n;i++)

cin>>a[i];

for(int i=1;i<=n;i++)

{

while(a[i]>1) //当前盒子巧克力总数大于1

{

x[++nx]=i; //记录当前多的一个巧克力所在位置 ++nx

a[i]--; //当前盒子的巧克力减1

}

if(a[i]==0) y[++ny]=i; //如果有空盒子 那么++ny

}

for(int i=1;i<=nx;i++)

for(int j=1;j<=ny;j++)

love[i][j]=-min(abs(x[i]-y[j]),n-abs(x[i]-y[j]));

//注意这里要取负数 求出最小的移动步数

cout<<-KM()<<endl;

}

return 0;

}

### 匈牙利算法

/\*

|求解最大匹配问题|

|递归实现|

\*/

vector<int>G[maxn];

bool inpath[maxn]; // 标记

int match[maxn]; // 记录匹配对象

void init()

{

memset(match, -1, sizeof(match));

for (int i = 0; i < maxn; ++i) {

G[i].clear();

}

}

bool findpath(int k) {

for (int i = 0; i < G[k].size(); ++i) {

int v = G[k][i];

if (!inpath[v]) {

inpath[v] = true;

if (match[v] == -1 || findpath(match[v])) { // 递归

match[v] = k; // 即匹配对象是“k妹子”的

return true;

}

}

}

return false;

}

/\*

|求解最大匹配问题|

|dfs实现|\*/

int v1, v2;

bool Map[501][501];

bool visit[501];

int link[501];

int result;

bool dfs(int x) {

for (int y = 1; y <= v2; ++y) {

if (Map[x][y] && !visit[y]) {

visit[y] = true;

if (link[y] == 0 || dfs(link[y])) {

link[y] = x;

return true;

} } }

return false;

}

void Search() {

for (int x = 1; x <= v1; x++) {

memset(visit,false,sizeof(visit));

if (dfs(x))

result++;

}

}

void hungary() {

int cnt = 0;

for (int i = 1; i <= m; i++) { // m为需要匹配的“妹子”数

memset(inpath, false, sizeof(inpath)); // 每次都要初始化

if (findpath(i)) cnt++;

}

cout << cnt << endl;

}

# 常用模板

### 常用头文件

#include <iostream>

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <time.h>

#include <stdlib.h>

#include <string>

#include <bitset>

#include <vector>

#include <set>

#include <map>

#include <queue>

#include <algorithm>

#include <sstream>

#include <stack>

#include <iomanip>

#define ll long long

#define ull unsigned long long

#define PI acos(-1.0)

#define eps 1e-12

#define fi first

#define se second

#define MEM(a,b) memset((a),(b),sizeof(a))

#define mod(x) ((x)%MOD)

#define pii pair<int,int>

#define wz cout<<"-----"<<endl;

const int INF\_INT = 2147483647;

const ll INF\_LL = 9223372036854775807LL;

const ull INF\_ULL = 18446744073709551615Ull;

const ll P = 92540646808111039LL;

const ll maxn = 1e5 + 10, MOD = 1e9 + 7;

const int INF=0x3f3f3f3f;

const int Move[4][2] = {-1,0,1,0,0,1,0,-1};

const int Move\_[8][2] = {-1,-1,-1,0,-1,1,0,-1,0,1,1,-1,1,0,1,1};

inline int read(){

int x=0,f=1;char ch=getchar();

while(ch<'0'||ch>'9'){if(ch=='-')f=-1;ch=getchar();}

while(ch>='0'&&ch<='9'){x=x\*10+ch-'0';ch=getchar();}

return x\*f;

}

### 读入写出挂

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

typedef unsigned long long ull;

typedef long double ld;

#define mp make\_pair

#define PI pair<int,int>

#define poly vector<ll>

#define For(i,l,r) for(int i=(int)(l);i<=(int)(r);i++)

#define Rep(i,r,l) for(int i=(int)(r);i>=(int)(l);i--)

#define pb push\_back

#define fi first

#define se second

inline char gc(){

static char buf[100000],\*p1=buf,\*p2=buf;

return p1==p2&&(p2=(p1=buf)+fread(buf,1,100000,stdin),p1==p2)?EOF:\*p1++;

}

#define gc getchar

inline ll read(){

ll x = 0; char ch = gc(); bool positive = 1;

for (; !isdigit(ch); ch = gc()) if (ch == '-') positive = 0;

for (; isdigit(ch); ch = gc()) x = x \* 10 + ch - '0';

return positive ? x : -x;

}

inline void write(ll a){

if(a<0){

a=-a; putchar('-');

}

if(a>=10)write(a/10);

putchar('0'+a%10);

}

inline void writeln(ll a){write(a); puts("");}

inline void wri(ll a){write(a); putchar(' ');}

inline ull rnd(){

return ((ull)rand()<<30^rand())<<4|rand()%4;

}

const int mod=998244353;

int ans=1;

int main(){

int h=read(),w=read();

For(i,1,h+w)ans=ans\*2%mod;

cout<<ans<<endl;

}

/\*

dp[j][i]=max(dp[j^1][k]+s[j][i]-s[j][k])

\*/

### 结构体重载运算符

//与平时cmp函数 相反

bool operator < (const node &p) const {

return r > p.r;

}

# 动态规划常见模型

### 最长上升子序列

/\*求LIS的长度\*/

/\*n方的写法，易写\*/

int ans = 0;

for(int i = 1; i <= n; ++i) {

scanf("%d", &num[i]);

dp[i] = 1;

for(int j = 1; j < i; ++j) {

if(num[j] <= num[i]) {

dp[i] = max(dp[i], dp[j] + 1);

}

}

ans = max(ans, dp[i]);

}

/\*手写二分，或者lower\_brond\*/

int binary(int x,int rr){

int l = 0,r = rr,ans;

while(l <= r){

int mid = l + r >> 1;

if(b[mid] >= x) ans = mid,r = mid - 1;

else l = mid + 1;

}

return ans;

}

int n;

for(int i = 1; i <= n;i++){

cin>>a[i];

}

int len = 1;

b[1] = a[1];

for(int i = 2;i <= n;i++){

if(a[i] > b[len]){

b[++len] = a[i];

}

else {

int t = binary(a[i],len);//也可以用c++自带的lower\_pound，自写也可以

b[t] = a[i];

}

}

cout<<len<<endl;

/\*

|最长上升子序列|

|状态转移|

|16/11/05ztx|

\*/

/\*

状态转移dp[i] = max{ 1.dp[j] + 1 }; j<i; a[j]<a[i];

d[i]是以i结尾的最长上升子序列

与i之前的 每个a[j]<a[i]的 j的位置的最长上升子序列+1后的值比较

\*/

void solve(){ // 参考挑战程序设计入门经典;

for(int i = 0; i < n; ++i){

dp[i] = 1;

for(int j = 0; j < i; ++j){

if(a[j] < a[i]){

dp[i] = max(dp[i], dp[j] + 1);

} } }

}

/\*

优化方法：

dp[i]表示长度为i+1的上升子序列的最末尾元素

找到第一个比dp末尾大的来代替

\*/

void solve() {

for (int i = 0; i < n; ++i){

dp[i] = INF;

}

for (int i = 0; i < n; ++i) {

\*lower\_bound(dp, dp + n, a[i]) = a[i]; // 返回一个指针

}

printf("%d\n", \*lower\_bound(dp, dp + n, INF) - dp;

}

/\*

函数lower\_bound()返回一个 iterator 它指向在[first,last)标记的有序序列中可以插入value，而不会破坏容器顺序的第一个位置，而这个位置标记了一个不小于value的值。

\*/

### 最长公共子序列

/\*

|求最长公共子序列|

|递推形式|

\*/

void solve() {

for (int i = 0; i < n; ++i) {

for (int j = 0; j < m; ++j) {

if (s1[i] == s2[j]) {

dp[i + 1][j + 1] = dp[i][j] + 1;

}else {

dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);

} } }

}

### 背包DP

//01背包

for(int i = 1;i <= n;i++){

for(int j = V;j >= w[i];j--){

dp[j] = max(dp[j],dp[j-w[i]]+v[i]);

}

}

//完全背包

for(int i = 0;i < n;i++){

for(int j = w[i];j <= e-s;j++){

dp[j] = min(dp[j],dp[j-w[i]]+v[i]);

}

}

//多重背包

for(int i = 0;i < n;i++){

for(int k = 0;k < b[i];k++){

for(int j = n;j >= v[i];j--){

dp[j] = max(dp[j],dp[j-v[i]]+w[i]);

}

}

}

//01背包 ，记录路径

#include<bits/stdc++.h>

using namespace std;

const int N = 1e5 + 10;

int pre[N],dp[N],v[N],ans[N];

void p(int x){

if(pre[x] == 0){

cout<<ans[x];return;

}

p(pre[x]);

cout<<' '<<ans[x];

}

int main(){

ios\_base::sync\_with\_stdio(0);

int n,m;cin>>n>>m;

for(int i = 0;i < n;i++)cin>>v[i];

for(int i = 0;i < N;i++)pre[i] = -1,dp[i] = -INF;

sort(v,v+n);

dp[0] = 0; //和01背包类似，因为是恰好装满，其他只要赋上负无穷

for(int i = 0;i < n;i++){

for(int j = m;j >= v[i];j--){

if(dp[j] <= dp[j-v[i]]+1){

dp[j] = dp[j-v[i]]+1;

ans[j] = v[i];

pre[j] = j - v[i];

}

}

}

if(dp[m] <= 0) cout<<"No Solution";

else p(m);

cout<<endl;

return 0;

}

# 线段树专题

### 线段树模板-区间更新&区间查询

10 5

1 2 3 4 5 6 7 8 9 10

Q 4 4

Q 1 10

Q 2 4

C 3 6 3

Q 2 4

4

55

9

15

#include<iostream>

#include<cstdio>

#include<cmath>

using namespace std;

const int maxn=100000+5;

const int INF=0x3f3f3f3f;

typedef long long LL;

struct node{

LL le,re;

LL mmax;

LL sum;

LL lazy;

LL len;

}tree[maxn<<2];

int t,k,q,n,m,a,b,c;

char ch[2];

LL val[maxn];

inline void pushup(int rt)

{

tree[rt].mmax=max(tree[rt<<1].mmax,tree[rt<<1|1].mmax);

tree[rt].sum=tree[rt<<1].sum+tree[rt<<1|1].sum;

}

inline void pushdown(int rt){

tree[rt<<1].lazy+=tree[rt].lazy;

tree[rt<<1|1].lazy+=tree[rt].lazy;

tree[rt<<1].mmax+=tree[rt].lazy;

tree[rt<<1|1].mmax+=tree[rt].lazy;

tree[rt<<1].sum+=tree[rt].lazy\*tree[rt<<1].len;

tree[rt<<1|1].sum+=tree[rt].lazy\*tree[rt<<1|1].len;

tree[rt].lazy=0;

}

//线段树建立

inline void build(int rt,int left,int right)

{

tree[rt].le=left;

tree[rt].re=right;

tree[rt].mmax=0;

tree[rt].lazy=0;

tree[rt].len=right-left+1;

if(left==right) {

tree[rt].sum=val[left];

return;

}

int m=(left+right)>>1;

build(rt<<1,left,m);

build(rt<<1|1,m+1,right);

pushup(rt);

}

inline void update(int rt,int le,int re,int lz){

if(tree[rt].le==le&&tree[rt].re==re){

tree[rt].mmax+=1;

tree[rt].sum+=tree[rt].len\*lz;

tree[rt].lazy+=lz;

return;

}

if(tree[rt].lazy) pushdown(rt);

int mid=(tree[rt].le+tree[rt].re)>>1;

if(le>mid) update(rt<<1|1,le,re,lz);

else if(re<=mid) update(rt<<1,le,re,lz);

else{

update(rt<<1,le,mid,lz);

update(rt<<1|1,mid+1,re,lz);

}

pushup(rt);

}

inline LL query(int rt,int le,int re){

if(tree[rt].le==le&&tree[rt].re==re)

return tree[rt].sum;

if(tree[rt].lazy) pushdown(rt);

int mid=(tree[rt].le+tree[rt].re)>>1;

if(le>mid) return query(rt<<1|1,le,re);

else if(re<=mid) return query(rt<<1,le,re);

else return query(rt<<1,le,mid)+query(rt<<1|1,mid+1,re);

}

int main()

{

while(~scanf("%d%d",&n,&m)){

for(int i=1;i<=n;i++)

scanf("%lld",&val[i]);

build(1,1,n);

for(int i=1;i<=m;i++){

scanf("%s",ch);

if(ch[0]=='Q'){

scanf("%d%d",&a,&b);

printf("%lld\n",query(1,a,b));

}

if(ch[0]=='C'){

scanf("%d%d%d",&a,&b,&c);

update(1,a,b,c);

}

}

}

return 0;

}

### 线段树模板-区间更新&最值查询

1 3 6 1 6 1 6 3 4 1 5 1 2 2 4

Case 1: 1 2 3 5

#include<bits/stdc++.h>

using namespace std;

const int maxn=1000000+5;

const int INF=0x3f3f3f3f;

typedef long long LL;

struct node{

int le,re;

int mmax;

int lazy;

}tree[maxn<<2];

int t,k,q,m,a,b;

char ch[2];

int ans[maxn];

inline void pushup(int rt)

{

tree[rt].mmax=max(tree[rt<<1].mmax,tree[rt<<1|1].mmax);

}

inline void pushdown(int rt){

tree[rt<<1].lazy+=tree[rt].lazy;

tree[rt<<1|1].lazy+=tree[rt].lazy;

tree[rt<<1].mmax+=tree[rt].lazy;

tree[rt<<1|1].mmax+=tree[rt].lazy;

tree[rt].lazy=0;

}

//线段树建立

inline void build(int rt,int left,int right)

{

tree[rt].le=left;

tree[rt].re=right;

tree[rt].mmax=0;

tree[rt].lazy=0;

if(left==right) {

//tree[rt].mmax=val[left];

return;

}

int m=(left+right)>>1;

build(rt<<1,left,m);

build(rt<<1|1,m+1,right);

pushup(rt);

}

inline void update(int rt,int le,int re){

if(tree[rt].le==le&&tree[rt].re==re){

tree[rt].mmax+=1;

tree[rt].lazy+=1;

return;

}

if(tree[rt].lazy) pushdown(rt);

int mid=(tree[rt].le+tree[rt].re)>>1;

if(le>mid) update(rt<<1|1,le,re);

else if(re<=mid) update(rt<<1,le,re);

else{

update(rt<<1,le,mid);

update(rt<<1|1,mid+1,re);

}

pushup(rt);

}

inline int query(int rt,int le,int re){

if(tree[rt].le==le&&tree[rt].re==re)

return tree[rt].mmax;

if(tree[rt].lazy) pushdown(rt);

int mid=(tree[rt].le+tree[rt].re)>>1;

if(le>mid) return query(rt<<1|1,le,re);

else if(re<=mid) return query(rt<<1,le,re);

else return max(query(rt<<1,le,mid),query(rt<<1|1,mid+1,re));

}

int main()

{

int cas=1;

scanf("%d",&t);

while(t--){

int len=0;

scanf("%d%d",&k,&q);

build(1,1,1000000 );

for(int i=1;i<=q;i++){

scanf("%d%d",&a,&b);

b--;

if(query(1,a,b)<k){

ans[len++]=i;

update(1,a,b);

}

}

printf("Case %d:\n",cas++);

for(int i=0;i<len;i++)

printf("%d ",ans[i]);

printf("\n\n");

}

return 0;

}

### 线段树模板-单点修改&区间查询

5 6 1 2 3 4 5 Q 1 5 U 3 6 Q 3 4 Q 4 5 U 2 9 Q 1 5

5 6 5 9

#include<bits/stdc++.h>

using namespace std;

const int maxn=2e5+5;

const int INF=0x3f3f3f3f;

typedef long long LL;

struct node{

int le,re;

int mmax;

}tree[maxn<<2];

int n,m,x,y;

char ch[2];

int val[maxn];

inline void pushup(int rt)

{

tree[rt].mmax=max(tree[rt<<1].mmax,tree[rt<<1|1].mmax);

}

inline void build(int rt,int left,int right)

{

tree[rt].le=left;

tree[rt].re=right;

if(left==right) {

tree[rt].mmax=val[left];

return;

}

int m=(left+right)>>1;

build(rt<<1,left,m);

build(rt<<1|1,m+1,right);

pushup(rt);

}

inline void update(int rt,int pos,int v){

if(tree[rt].le==tree[rt].re){

tree[rt].mmax=v;

return;

}

int mid=(tree[rt].le+tree[rt].re)>>1;

if(pos<=mid) update(rt<<1,pos,v);

else update(rt<<1|1,pos,v);

pushup(rt);

}

inline int query(int rt,int v)

{

if(tree[rt].le==tree[rt].re)

return tree[rt].le;

if(tree[rt<<1].mmax>=v) return query(rt<<1,v);

else return query(rt<<1|1,v);

}

inline int query\_max(int rt,int le,int re){

if(tree[rt].le==le&&tree[rt].re==re)

return tree[rt].mmax;

int mid=(tree[rt].le+tree[rt].re)>>1;

if(re<=mid) return query\_max(rt<<1,le,re);

if(le>mid) return query\_max(rt<<1|1,le,re);

return max(query\_max(rt<<1,le,mid),query\_max(rt<<1|1,mid+1,re));

}

int main()

{

while(~scanf("%d%d",&n,&m)){

for(int i=1;i<=n;i++)

scanf("%d",&val[i]);

build(1,1,n);

for(int i=1;i<=m;i++){

scanf("%s%d%d",ch,&x,&y);

if(ch[0]=='Q') printf("%d\n",query\_max(1,x,y));

if(ch[0]=='U') update(1,x,y);

}

}

return 0;

}